UAB

Treball de Fi de Grau

Grau en Enginyeria de Sistemes de Telecomunicació

Aplicación de Machine Learning para la Predicción Temporal en un Caso de Marketing

Javier Colin Pérez

Director: José López Vicario

Departament de Telecomunicació i Enginyeria de Sistemes

Universitat Autònoma de Barcelona

Bellaterra, Juny 19, 2018

UAB

El tribunal d'avaluació d'aquest Treball Fi de Grau, reunit el dia _____, ha acordat concedir la següent qualificació:

President: _____

Vocal: _____

Secretari: _____

UAB

El sotasignant, José López Vicario, Professor de l'Escola d'Enginyeria de la Universitat Autònoma de Barcelona (UAB).

CERTIFICA:

Que el projecte presentat en aquesta memòria de Treball Fi de Grau ha estat realitzat sota la seva direcció per l'alumne Javier Colin Pérez.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, 3 de Juliol de 2018.

Signatura:

Resum:

L'àrea del *Machine Learning* ha impactat a la societat fa pocs anys de forma massiva gràcies a la capacitat de computació i l'arribada del Big Data. Avui dia, ens trobem en una nova era en la que estem experimentant amb sistemes capaços de raonar i treure valor a aquestes quantitats ingents de dades per generar automàticament les seves hipòtesis. *Deep Learning*, un subcamp del *Machine Learning*, és el principal responsable d'això.

Aquest projecte persegueix l'objectiu d'analitzar diverses tècniques de predicció de dades, contrastant els avantatges i inconvenients de cadascuna. El procés de creació de diferents models permetrà afinar el resultat, obtenint així l'algoritme òptim.

Resumen:

El área del *Machine Learning* ha impactado a la sociedad hace pocos años de forma masiva gracias a la capacidad de computación y la llegada del Big Data. Hoy en día, nos encontramos en una nueva era en la que estamos experimentando con sistemas capaces de razonar y sacar valor a estas cantidades ingentes de datos para generar automáticamente sus hipótesis. *Deep Learning*, un subcampo del *Machine Learning*, es el principal responsable de esto.

Este proyecto persigue el objetivo de analizar diversas técnicas de predicción de datos, contrastando las ventajas e inconvenientes de cada una. El proceso de creación de diferentes modelos permitirá afinar el resultado, obteniendo así el algoritmo óptimo.

Summary:

The Machine Learning area has impacted society a few years ago in a massive way thanks to the computing capacity and the arrival of Big Data. Today, we are in a new era in which we are experimenting with systems capable of reasoning and extracting value from these huge amounts of data to automatically generate their hypotheses. Deep Learning, a subfield of Machine Learning, is primarily responsible for this.

This project aims to analyze various data prediction techniques, contrasting the advantages and disadvantages of each. The process of creating different models will refine the result, thus obtaining the optimal algorithm.

Índice

Listado de Figuras

Capítulo 1

Introducción

Los datos masivos están presentes cada vez más en nuestras vidas, y sin embargo apenas nos damos cuenta de sus aplicaciones. El desarrollo constante de la computación permite trabajar con estas cantidades de datos dando información y resultados relevantes si se tratan correctamente.

Los algoritmos de *Machine Learning* se aprovechan de la llegada del Big Data creando sistemas que aprenden automáticamente. Dicho en otras palabras, sistemas que aprenden mediante ejemplos.

La predicción de la demanda de un producto puede ser algo muy aleatorio para muchos negocios, sin embargo, con un buen histórico, podemos ajustarnos más gracias a esa experiencia. En este proyecto se trata de automatizar este proceso, analizando un conjunto de datos a partir de diferentes modelos de *Machine Learning* para la predicción.

1.1 Motivación del Proyecto

Conocer la futura demanda para las empresas e industrias es algo muy valioso a la hora de gestionar los recursos y stock [1]. Sobre todo, para aquellas en las que los productos tienen cierta caducidad; pues encargos excesivos pueden suponer grandes pérdidas para la compañía. Por otro lado, tampoco es conveniente hacer pedidos minimalistas ya que la empresa podría quedarse sin recursos y esto llevaría a la pérdida de clientes.

Podemos notar entonces que dichas empresas se encuentran en un debate constante, donde cada día del año puede ser diferente e inesperado. Con la ayuda de recopilar observaciones anteriores, se puede obtener una idea de lo que puede ocurrir en un futuro, no obstante, el ser humano no es capaz de tener en cuenta grandes cantidades de datos y procesarlas.

Es por ello que en este proyecto hacemos uso de la inteligencia artificial, sacando partido a estas bases de datos y a los recursos de computación actuales. Hace apenas unos años sonaría a ciencia ficción el alcance de estos algoritmos, sin embargo, hoy en día se están extendiendo creando empresas cada vez más competitivas.

Este éxito me cautivó y ha sido uno de los principales incentivos a la hora de llevar a cabo este trabajo. Actualmente y en los años venideros, se está viendo como las grandes empresas que disponen de grandes volúmenes de datos se están peleando por fichar a profesionales que sepan de *Machine Learning*, con lo cual, este primer contacto con el sector abre muchas puertas.

1.2 Objetivos

El principal objetivo de este proyecto es demostrar el rendimiento y verificar las prestaciones de algunos algoritmos de *Machine Learning* para la predicción, incluidas arquitecturas de redes neuronales.

En pocas palabras, nos centraremos en el análisis de una base de datos facilitada por el Departamento de Empresa de la UAB, en concreto, de *Kantar Worldpanel*. Hago un pequeño inciso y aprovecho esta mención para agradecer desde aquí haber compartido la información tanto a *Kantar Wordpanel* como al Departamento de Empresa de la UAB (concretamente del área de Marketing) para llevar a cabo este proyecto.

Dicha base de datos, trata a grandes rasgos sobre las ventas de un producto: yogures. Por lo tanto, gracias a estos datos de referencia se intentarán ajustar los algoritmos empleados de tal forma que sean capaces de tomar decisiones y reaccionar ante los cambios. A su vez, a partir de las diversas pruebas y contrastes entre modelos, se pretende ver que algoritmo es el más conveniente aplicar para nuestros datos y el que ofrece un mejor resultado.

1.3 Planificación del Proyecto

En esta sección se detallan las tareas por orden de ejecución que se han llevado a cabo para cumplir los objetivos propuestos en este proyecto. Son las siguientes:

1. **Formación**: Esencial para adquirir la base de nuevos conocimientos que no se han tratado previamente. Concretamente para familiarizarse con conceptos de *Machine Learning*, *Deep Learning* y de la Inteligencia Artificial en general.

   Por otro lado, para desarrollar el contenido de este proyecto también ha sido necesario dedicarle un entreno a Python, el lenguaje de programación más popular para proyectos de *Data Science* y el usado en este en concreto.

2. **Instalación del entorno de trabajo**: Como se ha mencionado en el paso 1, el lenguaje de programación usado es Python (v3.6.2). Este, se ha procesado en Anaconda (v1.6.2); un software para el procesamiento de datos a gran escala que incluye diferentes paquetes para ello. Recalcar también la instalación de la biblioteca TensorFlow (v1.2.1) en Anaconda, ya que ha sido el entorno que ha permitido trabajar con arquitecturas de redes neuronales.

3. **Preprocesado de datos**: Consiste en la preparación de los datos para que el proceso de minería de datos sea más fácil y efectivo. Para este proyecto se puede dividir en dos etapas. El primer proceso fue llevado a cabo por el tutor y el Departamento de Empresa de la UAB. Un primer proceso de limpieza de datos y eliminación de características redundante para poder construir el conjunto de datos (*data set*) principal. Y un segundo proceso para filtrar y montar los *data sets* a nuestra necesidad para trabajar con la información que realmente nos interesa.

4. **Desarrollo de los diferentes modelos y pruebas de test**: El núcleo del proyecto, la etapa que ha llevado más tiempo y donde se han creado los diferentes modelos de predicción. Los modelos empleados en este proyecto se detallan a continuación:

    - Regresor Lineal
    - Árbol de Decisión
    - Perceptrón Multicapa
    - Red Neuronal Recurrente
    - Red Neuronal Recurrente Combinada

    Esta tarea incluye el proceso de afinar los modelos; una evolución que requiere paciencia y, sobre todo, tiempo.

5. **Recolección de resultados y elaboración de la memoria**: Última etapa de este trabajo. Consiste en recolectar la información útil obtenida en el punto 4, detallarla y sacar las conclusiones. Para asentar mejor estos conocimientos también se dedica una pequeña introducción teórica para cada caso o concepto nuevo.

Para una fácil y cómoda visualización de las acciones, a continuación, se muestra la planificación y programación de las tareas llevadas a cabo a lo largo de este proyecto mediante un diagrama de Gantt.

Id	Nombre de tarea	Duración	Comienzo	Fin
1	**TFG**	**198 días**	**mié 20/09/17**	**vie 22/06/18**
2	**Formación**	**40 días**	**mié 20/09/17**	**mar 14/11/17**
3	Tutorial Python	15 días	mié 20/09/17	mar 10/10/17
4	Tutorial TensorFlow	13 días	mié 11/10/17	vie 27/10/17
5	Tutorial Deep Learning	12 días	lun 30/10/17	mar 14/11/17
6	**Preprocesado de datos**	**25 días**	**vie 22/09/17**	**jue 26/10/17**
7	**Instalación entorno de trabajo**	**2 días**	**lun 20/11/17**	**mar 21/11/17**
8	**Construcción data sets**	**6 días**	**mié 22/11/17**	**mié 29/11/17**
9	**Desarrollo de los modelos**	**92 días**	**jue 30/11/17**	**vie 06/04/18**
10	**Regresor Lineal**	**10 días**	**jue 30/11/17**	**mié 13/12/17**
11	Creación	6 días	jue 30/11/17	jue 07/12/17
12	Pruebas de test	4 días	vie 08/12/17	mié 13/12/17
13	**Árbol de Decisión**	**5 días**	**jue 14/12/17**	**mié 20/12/17**
14	Creación	2 días	jue 14/12/17	vie 15/12/17
15	Pruebas de test	3 días	lun 18/12/17	mié 20/12/17
16	**Perceptrón Multicapa**	**15 días**	**lun 08/01/18**	**vie 26/01/18**
17	Creación	10 días	lun 08/01/18	vie 19/01/18
18	Pruebas de test	5 días	lun 22/01/18	vie 26/01/18
19	**Red Neuronal Recurrente**	**21 días**	**lun 29/01/18**	**lun 26/02/18**
20	Creación	11 días	lun 29/01/18	lun 12/02/18
21	Pruebas de test	6 días	lun 19/02/18	lun 26/02/18
22	**Red Neuronal Recurrente Comb.**	**25 días**	**lun 05/03/18**	**vie 06/04/18**
23	Creación	15 días	lun 05/03/18	vie 23/03/18
24	Pruebas de test	5 días	lun 02/04/18	vie 06/04/18
25	**Recolección de resultados**	**26 días**	**lun 09/04/18**	**lun 14/05/18**
26	**Elaboración memoria TFG**	**25 días**	**lun 21/05/18**	**vie 22/06/18**

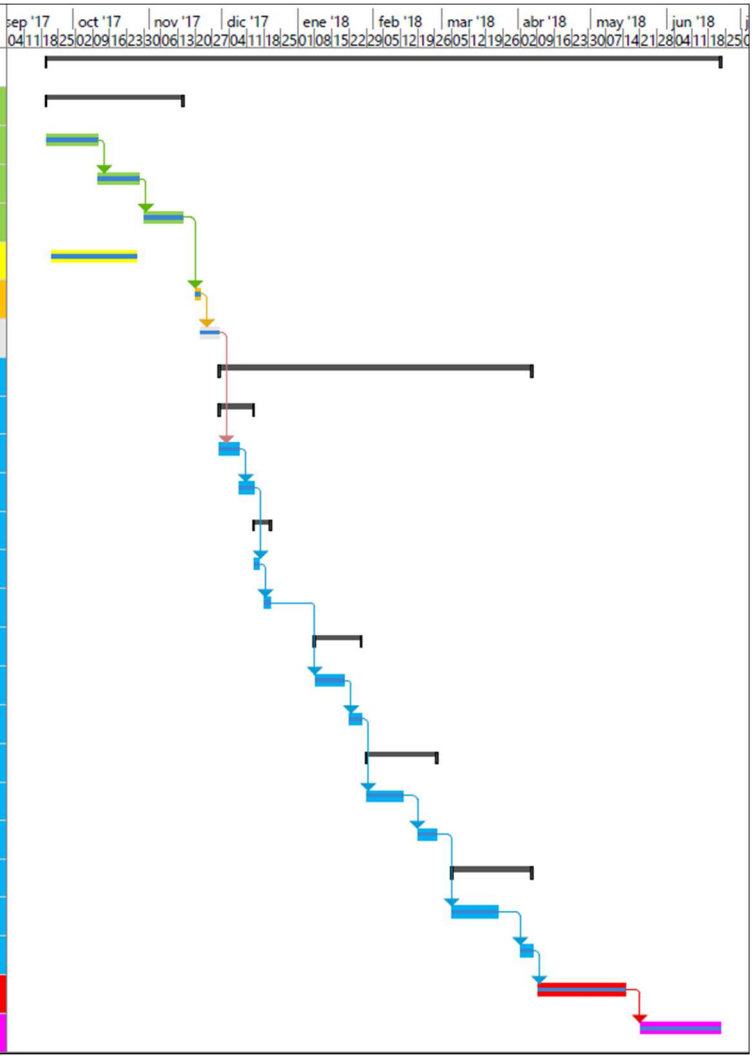


Figura 1: Diagrama de Gantt

Capítulo 2

Fundamentos de Machine Learning

En este capítulo se contemplan los conceptos básicos utilizados en este proyecto, esto engloba ver y conocer la convergencia del aprendizaje automático y el marketing hasta el día de hoy. Las técnicas de *Machine Learning* se utilizan como alternativa para resolver muchos problemas que el ser humano no es capaz de procesar. Aquí, se describe el alcance y el potencial que ofrece esta tecnología, en concreto, aplicada a casos reales de marketing, donde ya se está observando un gran impacto industrial.

2.1 Introducción al Machine Learning

La Inteligencia Artificial es un campo de estudio que busca emular el comportamiento inteligente en términos de procesos computacionales. Incluye *Machine Learning*, y es la disciplina científica capaz de crear sistemas que se adaptan a través de la experiencia, es decir, de reconocer los errores y rectificarlos sin estar explícitamente programados. Cada modelo está determinado por una función de coste proporcionada por el programador y la tarea del algoritmo es encontrar el valor que minimice la función de coste. El objetivo entonces es imitar la coincidencia de patrones que realizan los cerebros humanos [2].

El *Machine Learning* no es nada nuevo, es un concepto que no había experimentado su gran auge hasta hace pocos años. La diferencia entre ahora y cuando se crearon los primeros modelos es que antes no se disponía de las cantidades de datos actuales, y cuanta más información se alimenta en los algoritmos, más precisos se vuelven. Por ejemplo, si a una computadora se le muestra una manzana y se le indica que es una manzana, al principio la puede identificar como un objeto redondo y construir un modelo que establezca que, si algo es redondo, es una manzana. En el momento que se le introduzca una naranja, la computadora será más inteligente ya que deberá tener en cuenta que si algo es redondo y rojo es una manzana. Por lo tanto, la computadora modifica continuamente el modelo en función de la información que recibe. Es por ello que esto, ha dado pasos agigantados con la llegada del Big Data.

Además, para que esto sea factible, se necesita la contribución de computadoras mucho más rápidas, que sean capaces trabajar con estos conjuntos de datos y profesionales que sepan usar toda esta potencia de cálculo para poder entrenar un modelo. Lo cual, también es algo de lo que antes no disponíamos.

A nivel general, los algoritmos de *Machine Learning* se agrupan en algoritmos de aprendizaje supervisado y algoritmos de aprendizaje no supervisado [2]. En el aprendizaje supervisado damos al algoritmo un conjunto de datos del que sabemos como debe ser la salida, y donde hay una relación entre la entrada y la salida. En este tipo de aprendizaje se identifican los problemas de regresión y clasificación. Como veremos en nuestro caso (enfocado a la predicción), el conjunto de datos se prepara de tal forma que se pueda aplicar un algoritmo de aprendizaje supervisado. De lo contrario, existe el aprendizaje no supervisado, el cual solo tiene información de la entrada del conjunto de datos con el que se trabaja.

A su vez, dentro del concepto de *Machine Learning*, existe una rama llamada *Deep Learning*. El *Deep Learning* es la disciplina que representa un mayor acercamiento al modo de funcionamiento del sistema nervioso humano, imitando las microarquitecturas formadas por neuronas especializadas para realizar tareas específicas. Esto ha permitido crear redes que obtienen características ocultas en los datos, por ejemplo, dar valores de predicción inesperados y diferentes a los que el ser humano hubiera creído. Este enfoque ha dado mejores resultados gracias a su potencial de crear estructuras capaces de adaptarse a nuevas entradas de información. Para tener una visión general, a continuación, podemos ver un diagrama de Venn que muestra como el *Deep Learning* es un tipo de aprendizaje de representación, que a su vez es una especie de *Machine Learning*, que se utiliza para muchos, pero no todos, enfoques de la IA [3]:

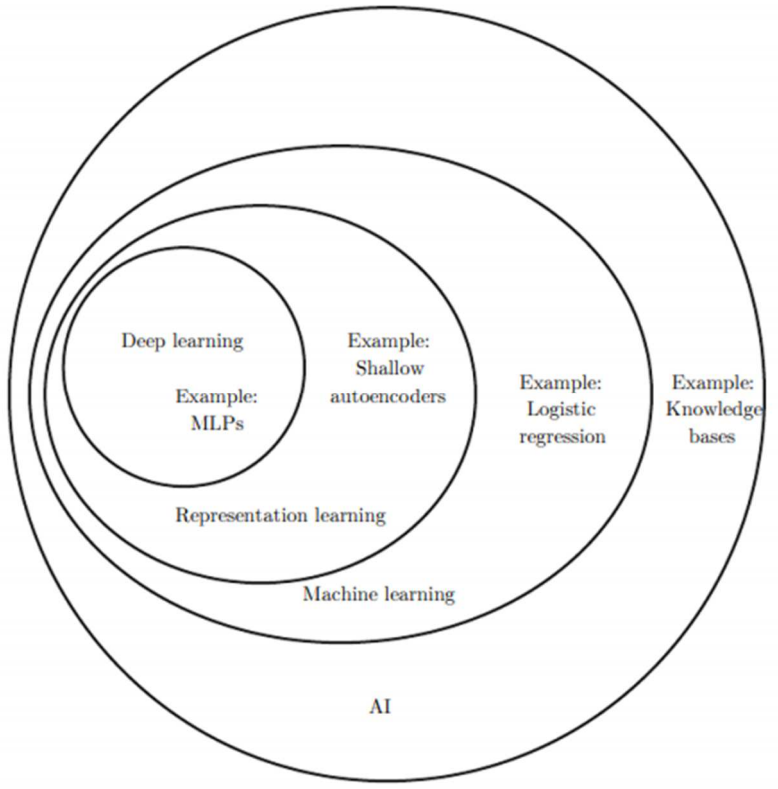


Figura 2: Diagrama de Venn para Inteligencia Artificial [3]

2.1.1 Redes Neuronales

Las redes neuronales son uno de los algoritmos más utilizados en *Machine Learning*. La idea básica detrás de una red neuronal es simular la interconexión de neuronas del ser humano dentro de una computadora para que pueda aprender cosas y tomar decisiones de manera similar. Es decir, son sistemas inspirados por las redes neuronales biológicas [3]. Su atractivo radica en que una red neuronal no necesita ser programada para aprender; aprende por si misma.

Esto se hace posible a partir de ejemplos. Un caso cotidiano de aprendizaje automático puede ser cuando hacemos una búsqueda en Google. En ese momento estamos haciendo uso de una de las mayores bases de datos del lenguaje actualizadas al instante. Por eso si escribimos con errores lo que estamos buscando, el algoritmo se da cuenta y nos pregunta si no quisimos decir otra cosa, colocando la palabra que cree que es correcta porque muchas personas la escribieron mal antes que nosotros.

Dichas redes están compuestas de neuronas (o unidades), organizadas en una serie de capas y conectadas entre ellas. Generalmente están formadas por tres partes; la capa de entrada, la(s) capa(s) intermedia(s) u oculta(s) (*hidden layer(s)*) y la capa de salida [4].

Las unidades de entrada están diseñadas para recibir diversa información que la red intentará reconocer o procesar. Por otro lado, las unidades de salida son las que responden a la información que la red ha aprendido. Entre dichas unidades, podemos encontrar una o más capas de unidades ocultas, que juntas forman la mayoría del cerebro artificial.

Las conexiones entre una unidad y otra están representadas por un número llamado peso, que puede ser positivo (si una unidad excita a otra) o negativo (si una unidad suprime o inhibe a otra) [5]. El peso aumentará o disminuirá según la intensidad de la señal en una conexión, es decir, cuanto mayor sea el peso, mayor será la influencia que una unidad tenga sobre otra. Es lo que se conoce como peso sináptico y se refiere a la fuerza de una conexión entre dos nodos.

Cada peso se va modificando en el llamado proceso de aprendizaje. La salida producida por una neurona viene dada por el sumatorio de cada entrada multiplicada por un peso de esa interconexión y puede escribirse como:

$$y = f\left(\sum \omega_i \cdot x_i\right)$$

A cada salida se le aplica una función de activación, que determina el estado de activación actual de la neurona. Las funciones de activación más usadas son la función

identidad, escalón, sigmoidea, sinusoidal… [5]. A continuación, se muestra un esquema del proceso explicado anteriormente para la salida de una neurona.

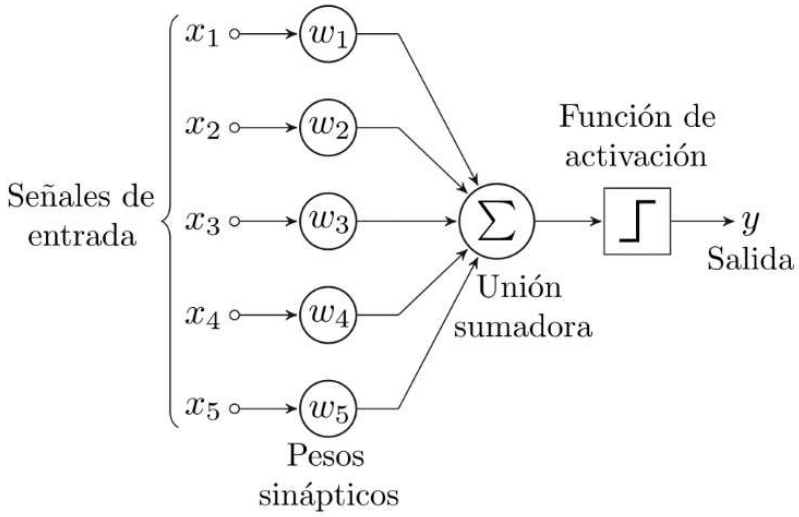


Figura 3: Modelo de neurona estándar [5]

Por lo tanto, a nivel general, para nuestro estudio tendremos la siguiente configuración:

- La capa de entrada: Serán las cantidades de yogures de los días anteriores al que queremos predecir. En esta capa no se genera ningún tipo de procesamiento, únicamente alimenta a la capa intermedia.

- La capa intermedia (*hidden layer*): Se encarga de procesar los datos que recibe sumando de forma ponderada los valores de entrada (cantidades de yogures) y aplicando una cierta función de activación. En dicha capa se le pueden aplicar tantas neuronas como se desee, aun que como veremos más adelante, más no implica mejor resultado. Y a su vez, tantas capas ocultas como queramos. Cuando la red está formada por dos o más capas ocultas se habla de *Deep Learning*.

- La capa de salida: Realiza el mismo cálculo que la capa oculta, pero utilizando la información de la capa anterior en lugar de la de la entrada. Para nuestro caso, la salida proporciona un vector que contiene las cantidades predichas según el algoritmo.

Para que la red neuronal aprenda, tiene que haber un elemento de reotroalimentación involucrado, del mismo modo que aprendemos los seres humanos cuando somos niños a distinguir lo que está bien y lo que esta mal. Es decir, recordamos lo que esta mal para corregir en la siguiente iteración y mejorar respecto al pasado. Las redes neuronales aprenden exactamente de la misma manera, por lo general mediante un proceso de retroalimentación llamado *backpropagation*. Esto implica comparar la salida que produce

una red con la salida que se pretendía producir, y usar la diferencia entre ellos para modificar los pesos de las conexiones entre las unidades en la red, trabajando desde las unidades de salida hasta las unidades de entrada (pasando por las unidades ocultas).

Para ello, es necesario definir una función de coste. Las funciones de coste (*error*, *loss* o *cost function*) se usan para estimar qué tan mal está funcionando el modelo. Es decir, nos sirve para saber si vamos mejorando en el aprendizaje de nuestra red y se estima ejecutando iterativamente el modelo, comparando las predicciones con los valores conocidos. Por lo tanto, el objetivo de un modelo *Machine Learning*, es encontrar una estructura que minimice lo máximo posible la función de coste.

En nuestro caso, la función de coste empleada es la raíz del error cuadrático medio (RMSE), la cual mide la diferencia entre los valores predichos por el modelo y los valores realmente observados. El principal motivo de uso de esta medida es porque es probablemente la medida estadística más fácil de interpretar, ya que tiene las mismas unidades que la de los valores que estamos comparando. Es decir, nos da la desviación del valor predicho respecto al real en términos de cantidad de yogures.

A nivel teórico, el RMSE mide cuánto error hay entre dos conjuntos de datos. En otras palabras, compara un valor predicho ($\hat{y}_k$) y un valor observado o conocido (y_k):

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y}_k)^2}$$

Y a nivel práctico, utilizando el mismo concepto, corresponde con la siguiente instrucción:

```
loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(y, outputs))))
```

Dónde:

- `y` = Pronósticos (valores esperados o resultados desconocidos)
- `outputs` = Valores observados (resultados conocidos)

Existen muchas otras alternativas para las funciones de coste, y no hay una mejor o peor, simplemente debemos utilizar la que mejor nos convenga para nuestro estudio. Algunas otras opciones podrían ser el *mean squared error* (MSE), *cross-entropy cost*, *exponential cost*, etc [5].

Con el tiempo, la retropropagación hace que la red aprenda, reduciendo la diferencia entre la salida real y la prevista hasta el punto en que los dos coinciden prácticamente, de modo que la red resuelve las cosas exactamente como debería. Para que esto suceda, también entra en juego otro factor muy importante, el optimizador de la función de

coste. Como ya sabemos, el valor que mejor se ajusta a nuestros datos es el que consigue un valor de error menor. Por lo tanto, si minimizamos esta función de coste o error, encontraremos el mejor modelo para nuestros datos.

El método responsable de esto es el Descenso del Gradiente (*Gradient Descent*), que permite minimizar una función eligiendo sistemáticamente valores de entrada y computando el valor de la función [6]. Dicho en otras palabras, es un algoritmo de optimización que intenta encontrar un mínimo local o global de una función.

De forma breve, el Descenso del Gradiente permite que el modelo aprenda la dirección que debe seguir para reducir errores (diferencias entre salidas reales y salidas predichas). A medida que el modelo itera, gradualmente converge hacia un mínimo, conocido como convergencia:

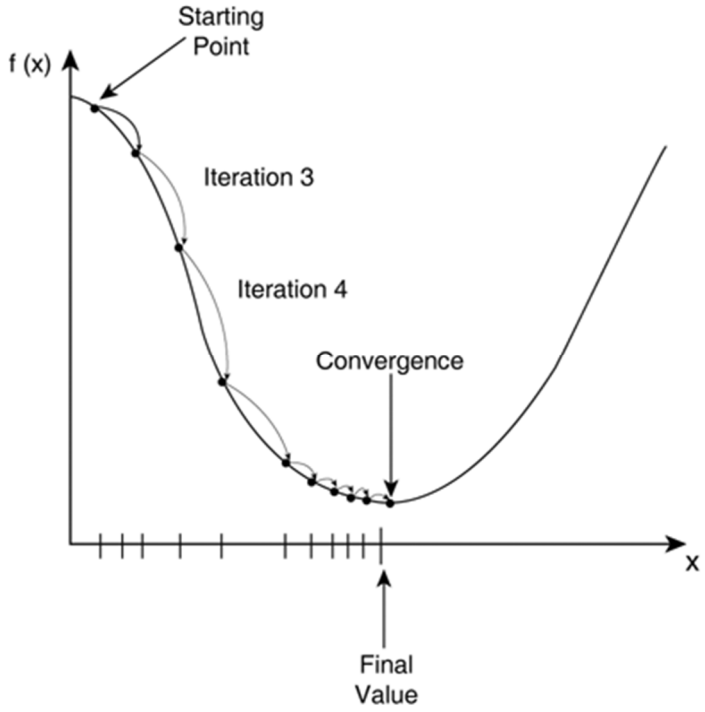


Figura 4: Función de ejemplo para el método del descenso de gradiente [6]

Como se puede ver, empieza con unos valores iniciales que iterativamente va modificando hasta acabar teniendo unos valores que minimicen la función de coste. El descenso del gradiente tiene un parámetro llamado velocidad de aprendizaje (*learning rate*). Como se puede ver arriba, inicialmente los pasos son más grandes, lo que significa que la tasa de aprendizaje es más alta y, a medida que el punto baja, la tasa de aprendizaje se hace más pequeña por el tamaño más corto de los pasos. En nuestro caso hemos empleado el optimizador Adam como algoritmo de optimización para el Descenso de Gradiente, pero existen más tipos: *Momentum*, *Adagradm*, *Nesterov accelerated gradient*, etc.

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

Después de cada pasada a través de la red neuronal en el entrenamiento, conocido como época (*epoch*), los pesos ω_i son ajustados y actualizados de tal manera que se va reduciendo el error.

Por lo tanto, la retropropagación o *backpropagation* incluye el proceso de aprender u operar desde que se recibe la información en la red hasta que se activan las capas ocultas y llegan a las unidades de salida (paso conocido como *feedforward*) y de calcular el error para volverlo a propagar a las capas anteriores y así actualizar los pesos.

A su vez, el proceso de aprendizaje implica dividir los datos en un conjunto de datos de entrenamiento y en un conjunto de datos de prueba. El conjunto de datos de entrenamiento sirve para dar ejemplos al algoritmo; en nuestro caso a partir de los días anteriores que sepa que cantidad de yogures van a haber el día que predice y que a partir de este entreno, pueda ajustar los parámetros (e.g. el peso entre las conexiones) del modelo. Recordemos que se trata de un aprendizaje supervisado, donde las salidas reales son conocidas y se puede ir corrigiendo el valor estimado respecto al valor real. Una vez entrenado el modelo con los datos de entrenamiento, se aplica sobre los datos de prueba, donde finalmente, es el conjunto que proporciona una evaluación del modelo después de haber sido entrenado. En este trabajo se ha establecido un conjunto de datos de entrenamiento del 80% y un conjunto de datos de prueba del 20% en todos los algoritmos para tener una comparación justa en el rendimiento de los diferentes modelos predictivos.

No obstante, a modo de análisis, también se ha optado por partir el conjunto de datos en tres partes: entreno 60% (*train*), validación 20% (*validation*) y prueba 20% (*test*). En el Capítulo 3 se detalla el uso y el fin de organizar los datos de esta manera.

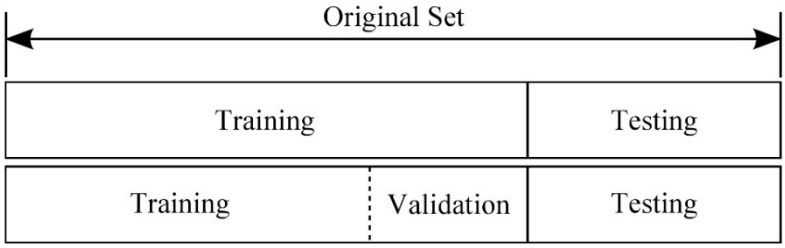


Figura 5: División del conjunto de datos original en dos o tres conjuntos [7]

En pocas palabras, cuando disponemos de tres conjuntos de datos, cada conjunto tiene una tarea específica y es la siguiente:

Training set – un conjunto de datos utilizado para aprender.

Validation set – un conjunto de datos utilizado para ajustar el modelo.

Testing set – un conjunto de datos utilizado para evaluar el rendimiento predictivo.

2.2 Estado del Arte

Durante estos últimos años, el sector de la inteligencia artificial está avanzando de manera vertiginosa por la repercusión que conlleva la llegada del Big Data de la mano de la potencia computacional actual. Con lo cual, se hace difícil definir un estado del arte para esta tecnología, pues en poco tiempo habrá exhibido un gran desarrollo. Sin embargo, mirando a nivel más general podemos asegurar que el subcampo que está experimentando gran interés es el de *Deep Learning*. En general, las redes neuronales, y esto es debido a la arquitectura/estructura que ofrecen los modelos, dando un comportamiento muy parecido al pensamiento de una persona.

Un modelo de *Machine Learning* ya entrenado permite grandes aplicaciones. Algunas de las más conocidas son la traducción automática de idiomas, sistemas de recuperación de información, producción automática de textos, reconocimiento de imágenes y voz, etc. Y esto, se logra empleando redes neuronales que procesan la información de entrada.

2.3 Redes Neuronales Aplicadas en Marketing

Los avances en la aplicación de la inteligencia artificial han llevado al desarrollo de sistemas que han demostrado ser útiles para los especialistas en marketing. Estos sistemas ayudan en áreas tales como la previsión de mercado, la automatización de procesos y la toma de decisiones, y aumentan la eficiencia de las tareas que normalmente realizarían los humanos. La ciencia detrás de estos sistemas se puede explicar a través de redes neuronales que procesan los datos de entrada y proporcionan resultados valiosos. Estas redes neuronales a través del proceso de aprendizaje, identifican relaciones y conexiones entre bases de datos. Una vez que se ha acumulado el conocimiento, se puede confiar en las redes neuronales para proporcionar generalizaciones y aplicar el conocimiento y el aprendizaje del pasado a una variedad de situaciones [3].

El gran impulso tecnológico que estamos viviendo ahora, ha revolucionado el entorno empresarial. Esta nueva era ha hecho que el mundo industrial almacene grandes cantidades de datos para previamente sacar valor de ellos. Es por ello que muchas organizaciones se ven obligadas a incorporar sistemas con inteligencia artificial, equivalentes a las de un ser humano, que sean capaces de procesar esos datos.

El reto que persiguen la mayoría de empresas es mejorar la comercialización de un producto, no obstante, cuando se habla de grandes cantidades es difícil gestionar el stock y la cantidad correcta para los distintos clientes. El principal motivo es porque estas cantidades fluctúan según la zona geográfica y el momento en el que nos encontremos.

El mundo industrial produce grandes niveles de datos a diario, sea del sector que sea. Recolectar o tener conciencia de estos datos es importante para que en un futuro podamos intuir la tendencia del mercado. En concreto, para aquellas empresas que desconocen la cantidad que van a vender de su producto y aún más todavía para aquellas en las que el producto que distribuyen tiene fecha de caducidad.

Pues bien, aquí entran en juego las redes neuronales, contribuyendo información a partir de los datos almacenados dando otro punto de vista valioso para la empresa. Las redes neuronales proporcionan una alternativa útil a los modelos estadísticos tradicionales debido a su fiabilidad, características de ahorro de tiempo y capacidad para reconocer patrones a partir de datos incompletos o ruidosos. Crear una red neuronal preparada para predecir la futura demanda es posible, y a medida que se alimente dicha red con mayor información de datos, más preciso se volverá el resultado [2]. No olvidemos que esto no es gratis, hay un compromiso entre precisión de los resultados y coste computacional. Lo cual implica que, si queremos mayor precisión, primero necesitaremos más datos de referencia y segundo, mayor potencia computacional.

No obstante, con un algoritmo bien ajustado a los datos que se dispone puede dar un mejor resultado y en menor tiempo del que podría dar el ser humano. Con lo cual, podemos notar que implementar técnicas de *Deep Learning*, ayudaría en gran medida a comercializar los productos para distintos clientes, donde cada uno, es diferente e imprevisible, tanto a nivel de demanda como en el momento que nos encontremos. Y como es lógico, conocer con antelación la cantidad que se debe preparar de un producto es realmente importante, ya que se gana tiempo en gestión y en gasto de inversión, lo cual supone un gran beneficio para la empresa.

Desde una perspectiva de marketing, las redes neuronales son una herramienta de software utilizada para ayudar en la toma de decisiones. A continuación, se muestran algunos ejemplos de aplicaciones que incluyen técnicas de *Deep Learning* en nuestro día a día, las cuales se aplican con un fin de negocio y marketing:

- Clasificación de patrones: Se pueden identificar los intereses de las personas a partir de ciertas palabras clave, proporcionando así, por ejemplo, información de los clientes que tienen más probabilidad de repetir una compra. Un ejemplo de caso sobre esto es el pronóstico con datos de Twitter; en este se compilaron y procesaron una gran cantidad de publicaciones de Twitter para extraer los sentimientos de la gente y poder averiguar sus intereses y pensamientos.

La Figura 6 intenta dar una visión general del proceso que implica este trabajo. Desde la transformación y el manejo de los datos en el origen hasta el resultado [8].

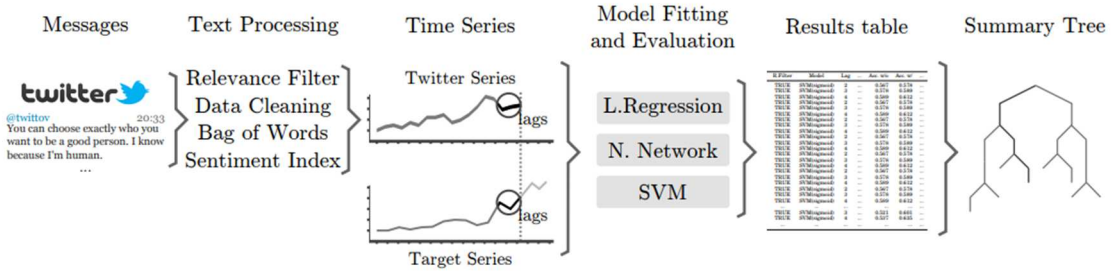


Figura 6: Visión general de los procesos de recopilación de datos, preprocesamiento, previsión y análisis final [8]

- Pronóstico: Son útiles para estimar eventos futuros, y tienen el objetivo de proporcionar puntos de referencia para monitorizar el rendimiento y reducir la incertidumbre. Facilitan el proceso de previsión, obteniendo más precisión en la demanda de productos y el control de inventario.

Un ejemplo de previsión utilizando redes neuronales es el "*Airline Marketing Assistant/Tactician*", una aplicación que permite la previsión de la demanda de pasajeros y la consecuente asignación de asientos en los aviones a través de redes neuronales [9].

- Análisis de marketing: Son las aplicaciones eficaces para extraer información estadística de grandes fuentes de datos. Un ejemplo de este enfoque es el "*Target Marketing System*", un sistema que escanea una base de datos de mercado e identifica clientes inactivos [9]. Esto permite tomar decisiones sobre a qué clientes clave dirigirse.

Como sabemos, las organizaciones se esfuerzan por satisfacer las necesidades de los clientes, prestando atención específica a sus deseos. Y como se ha introducido, las redes neuronales permiten comprender el comportamiento del consumidor, cosa que ayuda al profesional de marketing a tomar las decisiones adecuadas.

2.4 TensorFlow

Hoy en día, existen numerosos entornos para desarrollar algoritmos *Deep Learning* como Theano, Caffe, PyTorch, MXNet, etc, y es realmente difícil decidir que *framework* es el más apropiado utilizar por el constante desarrollo de estos. No obstante, por el momento, TensorFlow de Google parece ser el más utilizado, en gran parte por la manera de construir, entrenar y ejecutar redes neuronales de manera muy ágil, y por ello lo usaremos en este proyecto. A continuación, podemos ver los resultados de un pequeño estudio que analiza cual es el entorno más popular a partir de los intereses de búsqueda durante los últimos años.

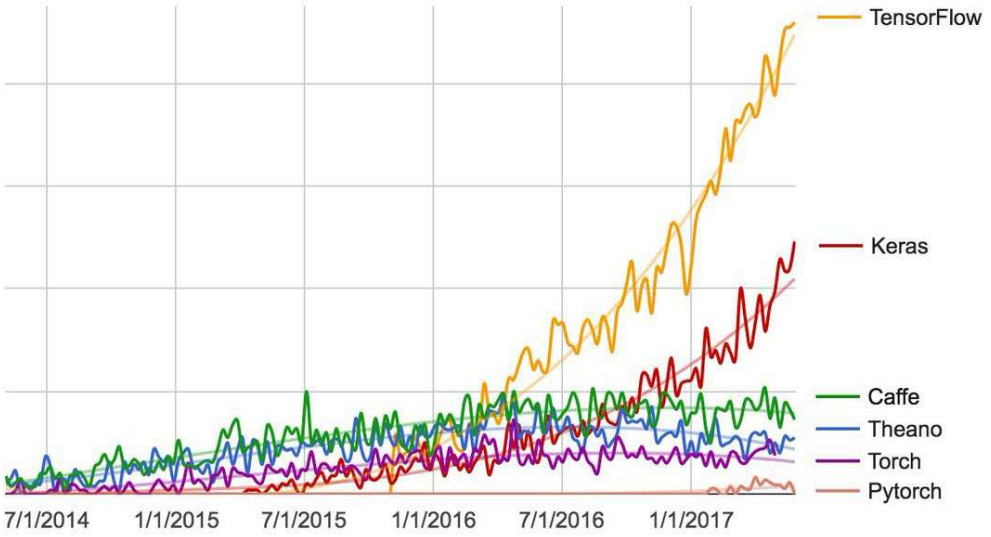


Figura 7: *Deep Learning framework search interest* [10]

Como se puede ver, desde que Google liberaba TensorFlow en 2015, esta tecnología ha resaltado en poco tiempo sobre los otras, siguiéndole Keras, una API de redes neuronales de alto nivel capaz de ejecutarse sobre TensorFlow.

A estas alturas podemos darnos cuenta que el paradigma del aprendizaje automático es difícil definirlo, pero para hacernos una idea, se muestra una tabla comparativa que analiza los *frameworks* más populares desde distintos puntos de vista [11]. Podemos notar entonces que cada entorno está construido de una manera diferente para diferentes propósitos, y es por ello que no existe uno mejor que otro; primero de todo debemos identificar el problema a resolver y a partir de ahí utilizar la herramienta que ofrezca un mejor resultado.

	Languages	Tutorials and training materials	CNN modeling capability	RNN modeling capability	Architecture: easy-to-use and modular front end	Speed	Multiple GPU support	Keras compatible
Theano	Python, C++	++	++	++	+	++	+	+
Tensor-Flow	Python	+++	+++	++	+++	++	++	+
Torch	Lua, Python (new)	+	+++	++	++	+++	++	
Caffe	C++	+	++		+	+	+	
MXNet	R, Python, Julia, Scala	++	++	+	++	++	+++	
Neon	Python	+	++	+	+	++	+	
CNTK	C++	+	+	+++	+	++	+	

Figura 8: Comparativa entre *frameworks* para *Deep Learning*

Actualmente, se puede decir que TensorFlow es uno de los mejores entornos para aprendizaje profundo y ha sido adoptado por varios gigantes como Airbus, Twitter, IBM y otros, principalmente por su arquitectura de sistema altamente flexible [11].

Y esta, es la principal diferencia que hace esta tecnología destacar frente a las otras. Si comparamos TensorFlow con el segundo entorno de trabajo más popular por el momento, Keras, vemos que Keras se trata de un lenguaje a más alto nivel y más amigable que TensorFlow por lo tanto nos preguntamos ¿por qué deberíamos usar TensorFlow para construir modelos de *Deep Learning*?. Pues bien, a veces, simplemente no se desea usar lo que ya está creado; se desea definir algo propio (E.g.: una función de coste, una capa, valor de *learning rate*, número de épocas de entreno, etc.). Y aunque Keras ha sido diseñado de tal manera que puede implementar casi todo lo que se desea, sabemos que las bibliotecas de bajo nivel ofrecen más flexibilidad. Es decir, los modelos se pueden ajustar en TensorFlow mucho más en comparación con Keras.

Por lo tanto, como se ha comentado anteriormente, antes de decidir que entorno utilizar debemos tener una idea del alcance que proporciona cada uno y del problema que queremos atacar. Siguiendo la comparativa anterior, si queremos construir y probar rápidamente una red neuronal con líneas de código mínimas probablemente elegiría Keras. Con Keras, se pueden construir redes neuronales en pocos minutos. Sin embargo, si se desea más control sobre la red y observar de cerca lo que sucede, TensorFlow es la opción correcta pese a su mayor complejidad.

Además, TensorFlow viene con una herramienta ampliamente utilizada y de gran utilidad: TensorBoard. Se utiliza para la visualización y monitorización del modelo a medida que avanza el aprendizaje; es una manera de controlar y saber lo que estamos haciendo. Es decir, los datos están escritos en TensorFlow y se pueden leer con Tensorboard.

Según Google, queda bastante por mejorar en este punto, pero se sabe que es un gran paso en la dirección correcta dado que actualmente, muchos modelos se ajustan a través de numerosas pruebas hasta dar con la mejor solución y disponer de una herramienta así ayudará en la creación de mejores modelos de manera más eficiente [12].

El caso de uso más conocido de TensorFlow es el Traductor de Google junto con capacidades tales como el procesamiento del lenguaje natural, clasificación/resumen de texto, reconocimiento de voz/imagen/escritura. No obstante, actualmente se continúan desarrollando aplicaciones mucho más inteligentes.

Llegados a este punto, una vez introducida la biblioteca que vamos a utilizar para desarrollar las redes neuronales, pasamos a ver como son los datos básicos que maneja TensorFlow y sus funciones. Evidentemente aquí se dan unas pinceladas de cómo se trabaja con esta biblioteca, entrar en detalle sería excesivo por las grandes cantidades de operaciones y expresiones que ofrece.

2.4.1 Conceptos de TensorFlow

En primer lugar y en pocas palabras, es importante saber que en TensorFlow se crea un gráfico computacional en el que se definen las constantes, variables y operaciones (objetos TensorFlow) para previamente ejecutarlo. Por lo tanto, el gráfico es una estructura de datos (*tensors*) que consta de todas las constantes, variables y operaciones que se desean hacer. Luego, cuando el modelo está listo, creamos una sesión (`tf.Session()`) de TensorFlow para posteriormente hacer la ejecución de la sesión. Esto ejecutará los nodos definidos y dará como salida los cálculos especificados.

Se conocen como tensores, a las unidades de datos que fluyen a través del modelo computacional de una red neuronal. Un tensor es cualquier conjunto de datos que puede tomar cualquier cantidad de dimensiones. Por lo tanto, es posible almacenar cualquier cantidad de atributos de manera muy organizada a partir de tensores.

Una estructura básica importante es el *placeholder*. Son un tipo de variables utilizadas para poder manipularlas durante la ejecución del programa, sin la necesidad de datos. Es decir, a este tipo de variables no se le asigna un valor específico, sino un valor que damos de entrada cuando le pidamos a TensorFlow ejecutar un cálculo.

En cuanto a operaciones, existen numerosas opciones para trabajar con las variables definidas, desde simples operaciones matemáticas hasta operaciones sobre matrices [10].

Pero antes cabe recordar que para poder empezar a utilizar todos los métodos y clases de TensorFlow, primero tenemos que importar la librería a nuestro archivo Python, y esto se consigue con el famoso comando:

```
import tensorflow as tf
```

A partir de este momento podemos empezar a especificar como será nuestro modelo.

También es importante mantener la información de los pesos y sesgos de nuestro modelo. Es lo que se conoce como *Variable*, y no es más que un tensor modificable que reside en el gráfico de operaciones. Dado que el peso de la neurona no será constante; esperamos

que cambie para aprender en función de la entrada y la salida "verdadera" que utilizamos para el entrenamiento.

Con el flujo de datos, el grafo se va describiendo formando un grafo de nodos y arcos. Los nodos típicamente implementan operaciones matemáticas, pero también pueden representar puntos de entrada de datos, salida de resultados, o lecturas/escritura de variables. Los arcos describen las relaciones entre nodos con sus entradas y salidas y transportan los tensores, la estructura de datos básica de TensorFlow [12]. Mencionar también que existe una función de activación para los nodos, la cual define la salida de cada nodo dada una entrada o un conjunto de entradas.

En resumen, se puede decir que la idea de programar en TensorFlow es:

1.- Especificar primero todo el problema.
2.- Crear una sesión que permite ejecutar la computación asociada.

Y que …
- Representa cálculos en forma de grafos.
- Ejecuta los grafos en el contexto de Sesiones.
- Representa los datos como tensores.
- Mantiene el estado con variables.
- Se alimenta de datos y devuelve los resultados de cada operación.

A continuación, podemos ver una pequeña muestra de lo que sería el cuerpo de una red neuronal definida usando TensorFlow.

```
# Una vez definidos nuestros datos (TRAIN/TEST), creamos nuestro gráfico TensorFlow que hará el cálculo:

tf.reset_default_graph()   # En caso de tener objetos gráficos anteriores en ejecución, este comando restablecería los gráficos

num_periods =      # Número de períodos que utilizamos para predecir
inputs =           # Número de vectores en la entrada
hidden =           # Número de neuronas. Se puede cambiar para mejorar la precisión
output =           # Número de vectores de salida

X = tf.placeholder(tf.float32, [None, num_periods, inputs])   # Creación de los objetos variables
y = tf.placeholder(tf.float32, [None, num_periods, output])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden, activation=tf.nn.relu)   # Crea nuestro objeto, en este caso RNN
rnn_output, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)            # Proporciona dos salidas, outputs y state

learning_rate =    # Define el valor de la tasa de aprendizaje

stacked_rnn_output = tf.reshape(rnn_output, [-1, hidden])          # Cambia la forma en un tensor
stacked_outputs = tf.layers.dense(stacked_rnn_output, output)      # Especifíca el tipo de capa (dense)
outputs = tf.reshape(stacked_outputs, [-1, num_periods, output])   # Reorganiza los resultados

loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(y, outputs))))    # Define la función de coste que evalúa el modelo
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)       # Método de optimización (Gradient Descent)
training_op = optimizer.minimize(loss)      # Entrena el resultado de haber minimizado la función de coste

init = tf.global_variables_initializer()    # Inicializa todas las variables

# Ahora ya podemos implementar este modelo en nuestros datos de entrenamiento. Especificamos el número de iteraciones/épocas
# que recorrerán nuestros datos, creamos nuestro objeto gráfico (tf.Session ()) e inicializamos nuestros datos para
# alimentarlos en el modelo a medida que recorremos las épocas.
```

Figura 9: Ejemplo de grafo en TensorFlow

Para este ejemplo se ha utilizado una red del tipo recurrente, pero como veremos más adelante existen más tipos. Siguiendo el ejemplo, se pueden identificar algunos conceptos mencionados anteriormente. Especificamos nuestros marcadores de posición variables. Inicializamos un tipo de celda RNN y el tipo de función de activación que queremos. ReLU significa 'Unidad Lineal Rectificada' y es la función de activación predeterminada, pero puede cambiarse a Sigmoid, Tangente hiperbólica (Tanh) y otras si se desea.

También se puede ver el hiperparámetro que define la tasa de aprendizaje llamado *learning rate*. En pocas palabras, es la tasa que marca la rapidez con que una red abandona viejas creencias por otras nuevas, o dicho de otra manera, la velocidad con la que aprende el algoritmo. Más adelante veremos como afecta el valor de este hiperpárametro y su definición con más detalle.

Seguidamente, reorganizamos las salidas ya que queremos que estén en el mismo formato que nuestras entradas para que podamos comparar nuestros resultados utilizando la función de pérdida (*loss*). Para nuestras redes, utilizamos la raíz del error cuadrático medio (RMSE), ya que este es un problema de regresión, en el cual nuestro objetivo es minimizar la diferencia entre lo real y lo predicho. Si estuviéramos lidiando con un resultado de clasificación, podríamos usar la entropía cruzada [12].

Y una vez definida la función de pérdida, es posible definir la operación de entrenamiento en TensorFlow que optimizará nuestra red de entradas y salidas. Para ejecutar la optimización, como hemos comentado anteriormente, utilizamos el optimizador de Adam; es un gran optimizador de uso general que realiza nuestro descenso de gradiente a través de una retropropagación en el tiempo. Esto permite una convergencia más rápida a costa de más computación [13].

Capítulo 3

Planteamiento del Problema

Llegados a este punto, es momento de aplicar los conceptos teóricos vistos anteriormente. En este capítulo se pretende demostrar las diferentes pruebas llevadas a cabo, viendo el flujo del porque se han ido aplicando los diferentes modelos predictivos y sus resultados.

Para ello es necesario volver la vista atrás y analizar este recorrido. Concretamente, se empieza haciendo una breve introducción del lenguaje de programación utilizado, para luego centrarnos en el procesado de datos, y finalmente, describir e implementar los modelos ya mencionados en el Capítulo 1.

Por lo tanto, este trabajo implica la integración de muchas técnicas, desde el modelado de datos hasta el análisis de los mismos, pasando por los diferentes modelos de *Machine Learning*. A continuación, se intenta dar una visión general de todas estas partes para que el lector pueda comprender e intuir las subsecciones que siguen.

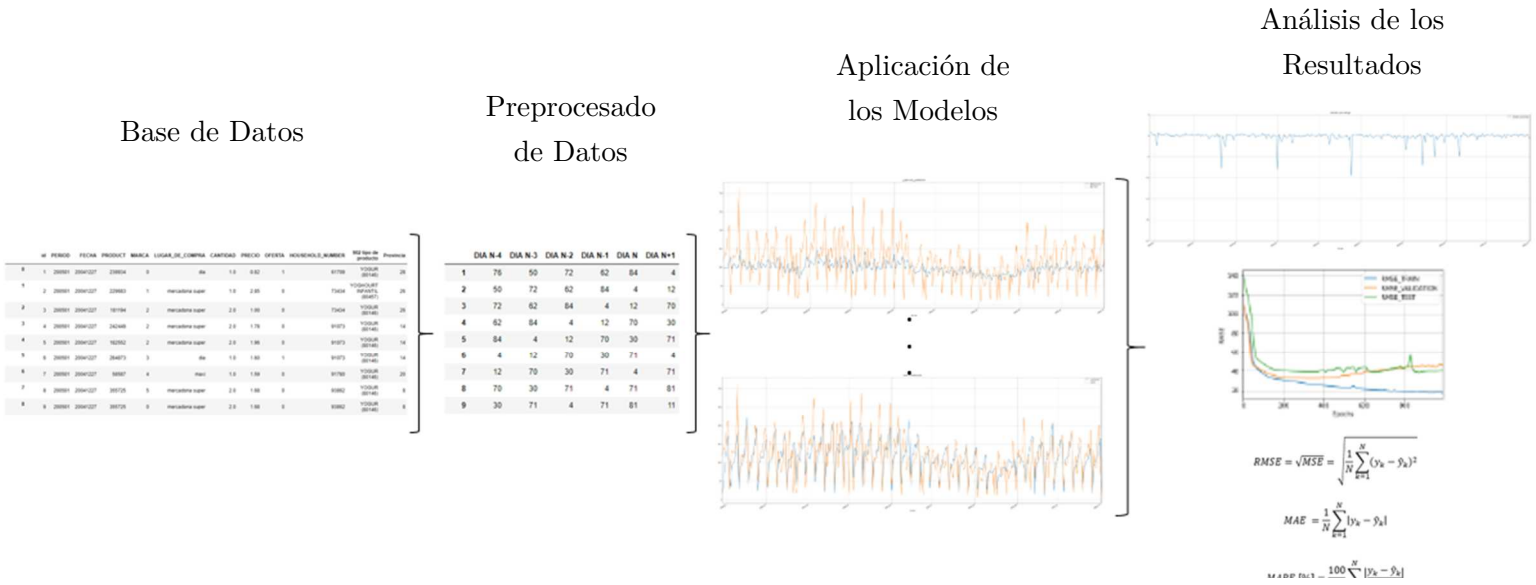


Figura 10: Esquema completo del contenido del Capítulo 3

Como podemos ver, partimos de una base de datos la cual contiene la información necesaria para desarrollar el proyecto. No obstante, antes de enfrentarnos al problema y aplicar modelos, es necesario trabajarla, obteniendo la información relevante y de la manera que nos interesa a partir de un preprocesado de dichos datos.

Una vez disponemos de los *data sets* como deseamos, ya podemos aplicar las diferentes técnicas de *Machine Learning* para la predicción. A posteriori, evaluaremos que tan buenos son estos modelos, y a su vez, comparándolos entre ellos a partir de diferentes técnicas y ecuaciones para el análisis predictivo.

3.1 Python para la Ciencia de Datos

Python es un lenguaje de programación poderoso y a la vez sencillo. En otras palabras, tiene un enfoque simple pero efectivo. No obstante, lo que de verdad lo hace atractivo para el análisis de datos es que dispone de numerosas librerías o módulos que permiten manipular datos de forma flexible, visual y eficiente [14].

Para este proyecto se hace uso de Pandas, un paquete que proporciona estructuras de datos, permitiendo trabajar de manera rápida e intuitiva. Podemos notar entonces, que el primer obstáculo para aplicar modelos de *Machine Learning* es adaptar los datos, dando la forma y el contenido deseado.

Otro punto fuerte de este lenguaje es que además de la capacidad de gestión y manejo de datos que proporciona, también incluye paquetes capaces de aplicar técnicas de *Machine Learning*. Es decir, existen bibliotecas que incluyen algoritmos de clasificación, regresión y agrupación entre otros.

En este proyecto empleamos Scikit-learn, una librería que proporciona varios métodos de fábrica de *Machine Learning*. La fácil implementación de estos modelos es un buen comienzo para poder realizar pruebas de forma muy simple. Sin embargo, más tarde veremos que dichas técnicas son poco precisas en los resultados, por lo que será necesario hacer uso de Tensorflow, la librería que permite desarrollar redes neuronales.

3.2 Descripción de los Datos Empleados

Como hemos visto, para todas las compañías de retail es fundamental ser capaces de anticiparse a la demanda para optimizar su logística y evitar la rotura de stock y pérdida de ventas.

Dado nuestro objetivo del proyecto; crear un algoritmo que prediga las ventas de un producto (yogures), empleamos diferentes modelos de predicción para ver cual se adapta mejor a lo esperado y así, ser capaces de anticiparnos a la demanda aplicando inteligencia a la información histórica de la que disponemos.

Para ello, como hemos introducido brevemente, empezaremos desde modelos ya definidos por librerías de Python hasta redes neuronales ajustadas a nuestros datos. Evidentemente, para conseguir una comparación justa entre modelos, usaremos la misma estructura del *data set* para todos. Un *data set* no es más que un conjunto de datos organizado y estructurado, y en nuestro caso, deberemos hacerlo de una manera específica.

Antes de todo, y según se ha comentado, es necesario hacer un preprocesado de los datos, para así quedarnos con los datos que nos interesan y de la forma que nos interesan. Como punto de partida, nuestro conjunto de datos tiene el siguiente aspecto:

	id	PERIOD	FECHA	PRODUCT	MARCA	LUGAR_DE_COMPRA	CANTIDAD	PRECIO	OFERTA	HOUSEHOLD_NUMBER	902 tipo de producto	Provincia
0	1	200501	20041227	239934	0	dia	1.0	0.82	1	61709	YOGUR (80146)	28
1	2	200501	20041227	229683	1	mercadona super	1.0	2.85	0	73434	YOGHOURT INFANTIL (80457)	26
2	3	200501	20041227	181194	2	mercadona super	2.0	1.00	0	73434	YOGUR (80146)	26
3	4	200501	20041227	242449	2	mercadona super	2.0	1.78	0	91073	YOGUR (80146)	14
4	5	200501	20041227	182552	2	mercadona super	2.0	1.96	0	91073	YOGUR (80146)	14
5	6	200501	20041227	284873	3	dia	1.0	1.60	1	91073	YOGUR (80146)	14

Figura 11: Base de datos inicial

Se trata de nuestra base de datos, la cual debemos pulir para tener la información útil. Esta estructura de datos se conoce como un *data frame*, y a diferencia de los *data sets*, puede contener distintos tipos de datos. En ella, se guardan las compras que llevan a cabo un conjunto de amas de casa, en concreto yogures. Podemos ver que cada entrada es una fecha donde se indica la compra realizada por una ama de casa, donde se pone además la marca, el lugar de compra, la provincia, la cantidad de yogures comprados, etc. Cada provincia se identifica por un número único, en nuestro caso, trabajamos con la provincia de Barcelona (#8), por lo tanto, podemos empezar a filtrar datos.

Código	Provincia	Código	Provincia
1	ALAVA	26	RIOJA
2	ALBACETE	27	LUGO
3	ALICANTE	28	MADRID
4	ALMERIA	29	MALAGA
5	AVILA	30	MURCIA
6	BADAJOZ	31	NAVARRA
7	BALEARES	32	ORENSE
8	BARCELONA	33	ASTURIAS
9	BURGOS	34	PALENCIA
10	CACERES	35	LAS PALMAS
11	CADIZ	36	PONTEVEDRA
12	CASTELLON	37	SALAMANCA
13	CIUDAD REAL	38	TENERIFE
14	CORDOBA	39	CANTABRIA
15	LA CORU¥A	40	SEGOVIA
16	CUENCA	41	SEVILLA
17	GERONA	42	SORIA
18	GRANADA	43	TARRAGONA
19	GUADALAJARA	44	TERUEL
20	GUIPUZCOA	45	TOLEDO
21	HUELVA	46	VALENCIA
22	HUESCA	47	VALLADOLID
23	JAEN	48	VIZCAYA
24	LEON	49	ZAMORA
25	LERIDA	50	ZARAGOZA

Figura 12: Código y Provincias del *Data Frame*

Además, como queremos unificar las compras para una única provincia (Barcelona), deberemos sumar todas las compras de yogures para los mismos días, obteniendo así el total por fechas. Una vez hecho esto, debemos asegurarnos que se mantienen en orden, y para una mejor visualización, cambiar el formato de fecha a por ejemplo yyyy-mm-dd.

Los primeros pasos entonces para modelar nuestro *data frame* son:

- Reducir el *data frame* seleccionando "Provincia" == 8 (Barcelona)
- Sumar todos los yogures de la misma fecha para obtener una única entrada
- Convertir la "FECHA" en formato leíble (E.g.: 20041227 a 2004-12-27)
- Organizar las compras en una serie temporal

En resumen, se genera un *data frame* donde solo habrá la columna "CANTIDAD", que contiene los yogures comprados para un día en concreto por todas las amas de casa de Barcelona desde el 27/12/2004 hasta el 02/11/2008 (≅ 4 años). Este es el resultado:

FECHA	CANTIDAD
2004-12-27	76
2004-12-28	50
2004-12-29	72
2004-12-30	62
2004-12-31	84
2005-01-01	4
2005-01-02	12
2005-01-03	70
2005-01-04	30
2005-01-05	71
2005-01-06	4

Figura 13: *Data Frame* de las cantidades por fecha

Hecho esto, debemos transformar nuestro *data frame* de tal forma que con datos pasados podamos predecir los siguientes, es decir, diferenciar por entradas y salidas. En concreto usamos una estructura tal que dado el tiempo actual (DIA N), así como los 4 días anteriores (DIA N-1), (DIA N-2), (DIA N-3) y (DIA N-4) predecir el valor de (DIA N+1). Por ejemplo, dado el *data frame* anterior:

Con las muestras X → 76, 50, 72, 62 y 84
Predecir la muestra Y → 4

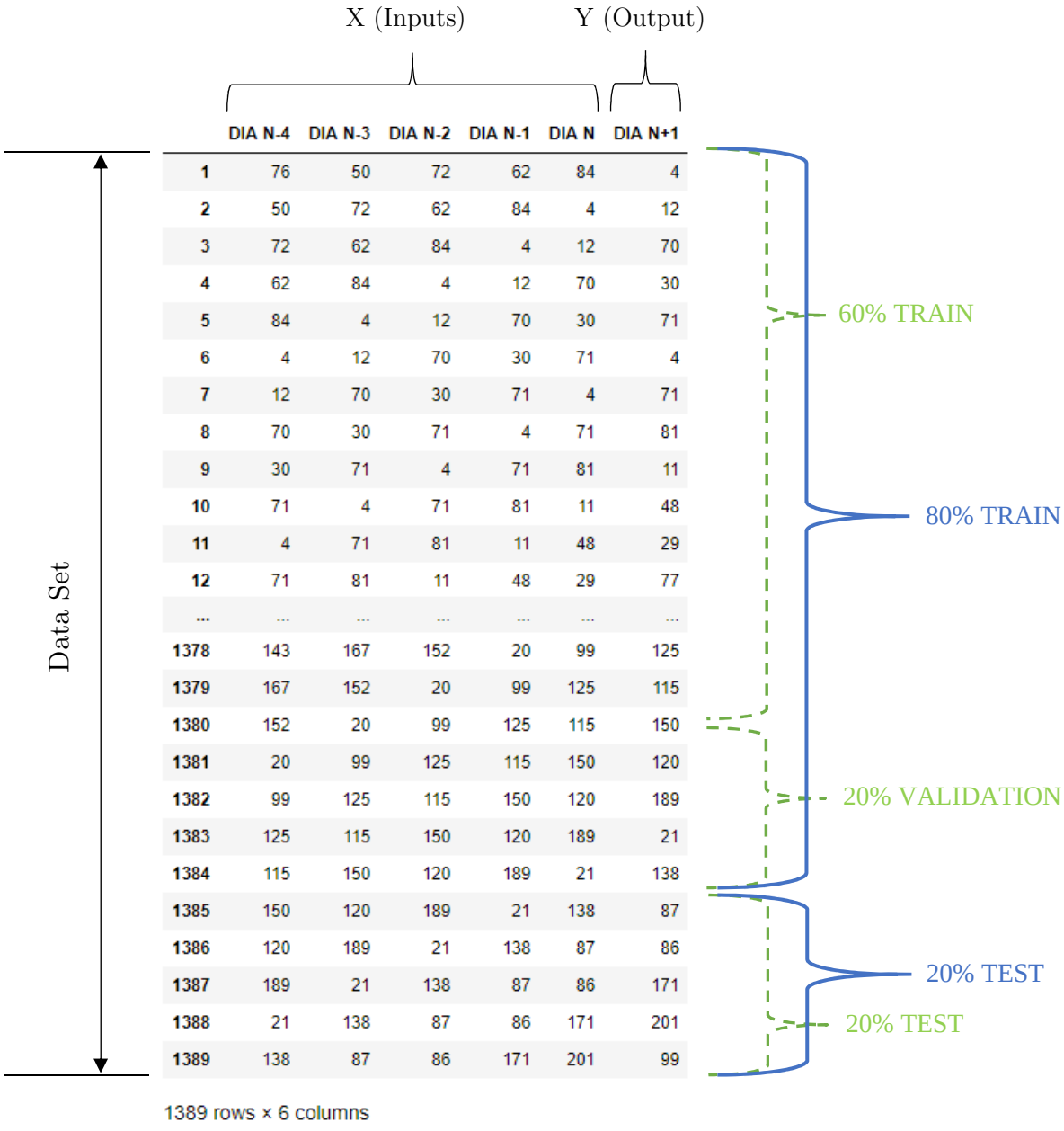
Con las muestras X → 50, 72, 62, 84 y 4
Predecir la muestra Y → 12

Con las muestras X → 72, 62, 84, 4 y 12
Predecir la muestra Y → 70

Etc.

A continuación, se puede contemplar el *data set* descrito. Como podemos ver, se trata de un modelo de aprendizaje supervisado, donde se cuenta con un conjunto de ejemplos de los cuáles conocemos la respuesta.

Para todos los modelos se establece el mismo porcentaje de *train* y de *test*, un 80% de *train* y 20% de *test*. A modo de ajuste y de análisis para afinar las redes neuronales, se divide el conjunto en tres partes: un 60% de *train*, un 20% de *validation* y un 20% de *test*. Trabajar con un subconjunto de datos más (*validation*) antes de aplicar el algoritmo sobre los datos de prueba, permite darnos ciertas pistas para evaluar y corregir nuestro modelo. Evitando así el sobreajuste o sobreentreno (*overfitting*) entre otros. En la sección de análisis, se entra en más detalle en este tema, pero por ahora es conveniente saber que existen estas dos posibilidades de división de datos:



	DIA N-4	DIA N-3	DIA N-2	DIA N-1	DIA N	DIA N+1
1	76	50	72	62	84	4
2	50	72	62	84	4	12
3	72	62	84	4	12	70
4	62	84	4	12	70	30
5	84	4	12	70	30	71
6	4	12	70	30	71	4
7	12	70	30	71	4	71
8	70	30	71	4	71	81
9	30	71	4	71	81	11
10	71	4	71	81	11	48
11	4	71	81	11	48	29
12	71	81	11	48	29	77
...	...	...	...	...	...	...
1378	143	167	152	20	99	125
1379	167	152	20	99	125	115
1380	152	20	99	125	115	150
1381	20	99	125	115	150	120
1382	99	125	115	150	120	189
1383	125	115	150	120	189	21
1384	115	150	120	189	21	138
1385	150	120	189	21	138	87
1386	120	189	21	138	87	86
1387	189	21	138	87	86	171
1388	21	138	87	86	171	201
1389	138	87	86	171	201	99

1389 rows × 6 columns

Figura 14: Divisiones del *data set*

3.3 Estudio de Eventos

Es importante también realizar un estudio de los eventos sobre el conjunto de datos que vamos a tratar. Esto nos permite contemplar de cerca el comportamiento de la secuencia a predecir y saber por qué el algoritmo de predicción ha fallado en ciertas fechas. Por ejemplo, a continuación, se han graficado las últimas 280 cantidades de yogures comprados en Barcelona y aparentemente parece una señal muy ruidosa, aleatoria y sin sentido. Sin embargo, analizando en detalle encontramos algo de periodicidad.

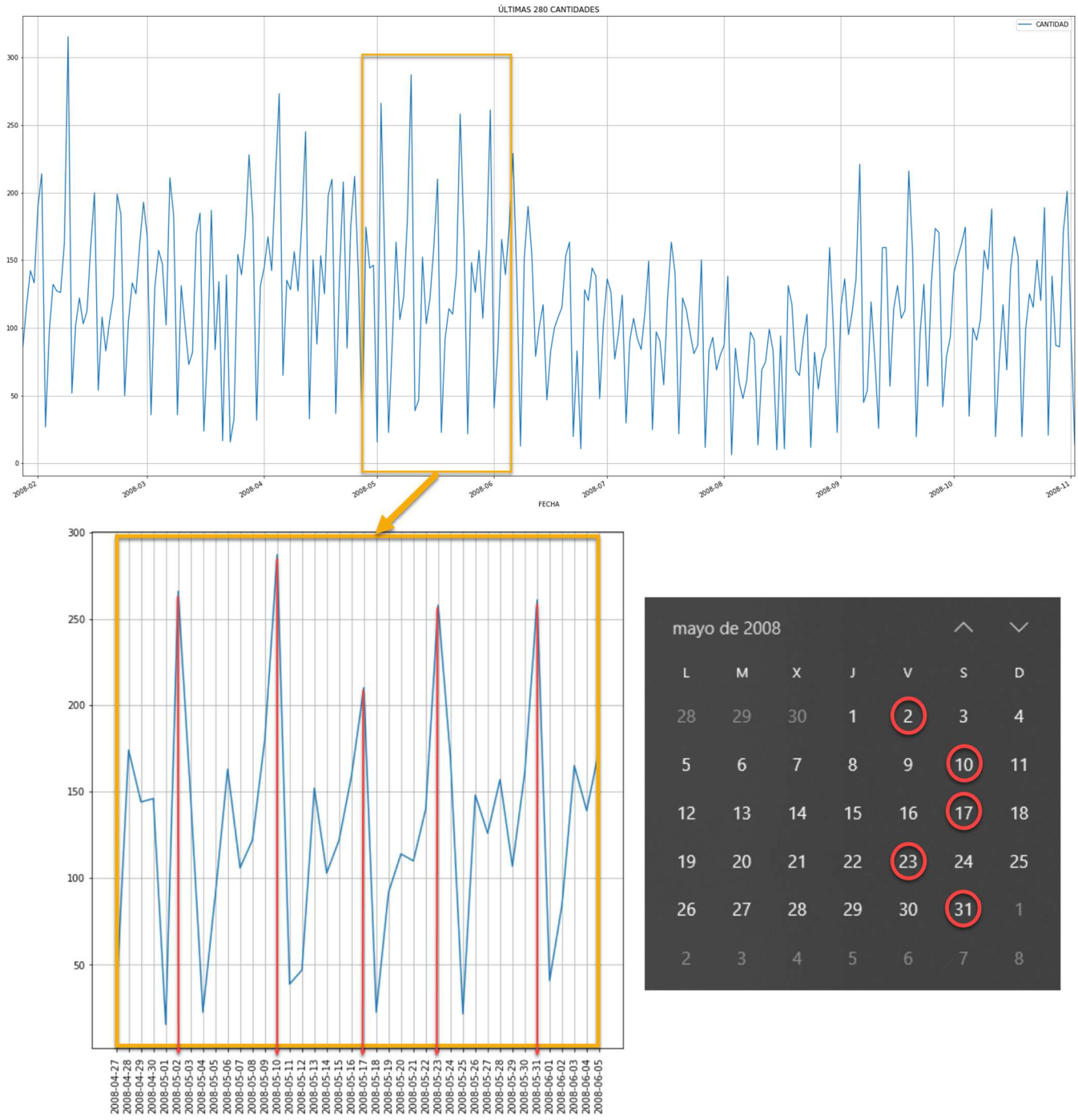


Figura 15: Tendencia de compra para el fin de semana

En la Figura 15 podemos ver en ese pequeño tramo (mayo del 2008) como se repite la misma secuencia de forma muy parecida para cada semana. Por propia experiencia y según el ejemplo de este estudio, la mayoría de nosotros realizamos las compras el fin de semana (hablo de viernes y sábado) para abastecernos de alimentos durante toda la semana siguiente, dejando entonces el domingo y el lunes como días menos indicados para hacer la compra. No obstante, como podemos observar existe un segundo pico el cual se mantiene más o menos uniforme para cada semana e indica que otra gran mayoría aun que aproximadamente la mitad, compra el martes.

En gran parte esto es bueno, ya que facilita el aprendizaje a las redes neuronales; sabrán que para el lunes la demanda de yogures será escasa y para el fin de semana (viernes y sábado en concreto), será alta. Pero por otro lado, esto tiene su inconveniente y un precio a pagar. Tener una secuencia muy repetitiva hace que la red neuronal no generalice bien, y es debido a que se acostumbra a recibir ciclos de datos muy parecidos. Es decir, ajustamos la red a unos datos de entrenamiento corriendo el riesgo de "sobreaprender" (*overfitting*). Se conoce como *overfitting* y se da cuando el modelo se ajusta demasiado a los datos observados y no generaliza bien, es decir en ese estado si la red recibe unos datos de entrada distintos a los que solía ver, probablemente fallará. A pesar de ello, existen estrategias para evitar el sobreaprendizaje como obtener más datos (la mejor opción si tenemos capacidad para entrenar la red), ajustar los parámetros de la red para que tenga la capacidad adecuada (suficiente para identificar las regularidades en los datos, pero no demasiada para identificar datos inusuales), combinar modelos, etc.

Volviendo a nuestro ejemplo, una buena herramienta para identificar donde la red ha fallado es el residuo porcentaje. En la siguiente sección se explica con más detalle, por ahora es suficiente saber que básicamente mide la diferencia entre el valor predicho y el real y si se grafica tiene el siguiente aspecto:

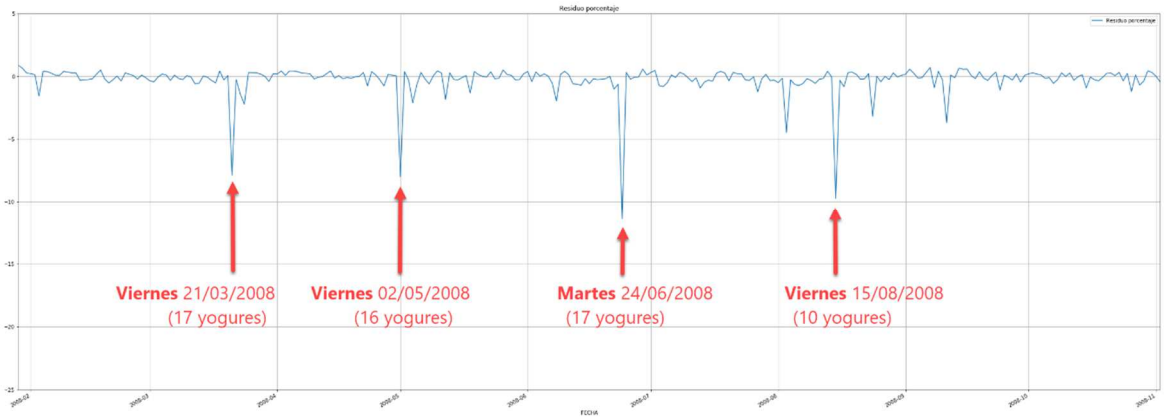


Figura 16: Eventos inesperados

Donde los picos más pronunciados corresponden con aquellas fechas en las que la red se esperaba un valor alto (viernes y martes) cuando en realidad la compra para esos días era de unos pocos yogures; valores inusuales que la red no ha sabido predecir.

3.4 Métodos de Análisis

Para cada uno de los escenarios que se generen, será necesario medir el rendimiento de ellos mediante algún indicador de precisión. Y es por ello que antes de empezar a ver cómo funciona el primer modelo, introduciremos como se mide la precisión de los pronósticos. En primer lugar, comentar que se tratan de pronósticos de valores ya pasados y conocidos (aprendizaje supervisado), y estas medidas no proporcionan una idea de la precisión del pronóstico en el futuro. Por lo tanto, es importante comprender que debemos suponer que el pronóstico para el futuro será tan preciso como lo ha sido en el pasado, y que por lo tanto garantiza la misma precisión para un pronóstico de futuro. Dicho esto, vamos a ver como se determina la precisión de los algoritmos.

Como "desarrollador" de modelos predictivos, el principal propósito es obtener el menor error entre el valor pronosticado y el valor real. Para analizar esto, existen múltiples herramientas. A continuación, se contemplan las utilizadas en este proyecto:

1. Gráfico de traza real vs traza predicha: Es la forma más rápida, cómoda y visual de ver si un algoritmo está trabajando bien o no. Es decir, si graficamos el vector y_k (valores observados) sobre el vector $\hat{y}_k$ (valores estimados) en función del tiempo podremos intuir como funciona; dependiendo de si dichas trazas están más solapadas o menos.

2. Gráfico del residuo: El residuo no es más que un vector que contiene las diferencias entre los valores observados (y_k) y los estimados ($\hat{y}_k$):

$$Residuo = y_k - \hat{y}_k$$

   Dibujar este vector, nos sirve entonces para ver la diferencia entre dichos valores. Por lo tanto, si el resultado es una señal plana próxima a 0 querrá decir que existe poca diferencia entre los valores reales y predichos, por lo que el algoritmo estará haciendo una buena predicción.

3. Gráfico del residuo porcentaje: El residuo porcentaje es muy similar al residuo, salvo que en este caso dividimos por el valor absoluto del vector de valores reales:

$$Residuo\ Porcentaje = (y_k - \hat{y}_k)/|y_k|$$

   Esto nos permite obtener un trazado donde los picos (positivos o negativos) nos "chivan" dónde ha habido mayor diferencia o error, es decir, dónde la red ha fallado. Para una mejor comprensión de esto, a continuación, se puede ver un ejemplo:

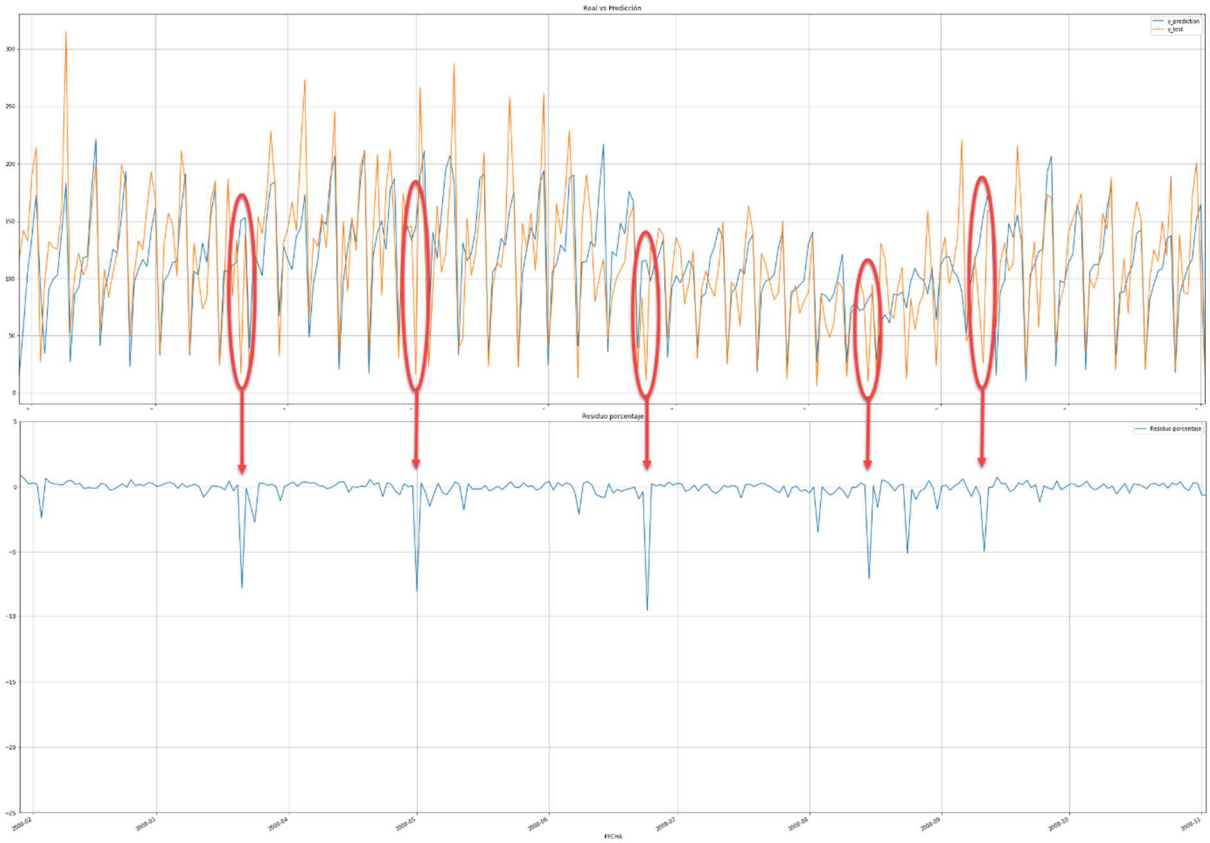


Figura 17: Ejemplo de análisis a partir del residuo porcentaje

4. Cálculo del RMSE: La raíz del error cuadrático medio o RMSE (*Root Mean Squared Error*) es una de las medidas más utilizadas para medir el error entre el valor real y el predicho. Esta medida es la raíz del promedio de los cuadrados del error para cada valor y se puede escribir de la siguiente manera:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y}_k)^2}$$

   Lo que realmente hace interesante usar esta medida es que el resultado está en unidades originales de la información histórica, es decir, para nuestro caso, cantidades de yogures. Evidentemente, cuanto mayor es el RMSE, más inconsistente es el algoritmo para predecir.

5. Cálculo del MAE: El error absoluto medio o MAE (*Mean Absolute Error*) es otra herramienta común para medir la precisión. Y de la misma forma que el RMSE, expresa el error de predicción del modelo.

$$MAE = \frac{1}{N}\sum_{k=1}^{N}|y_k - \hat{y}_k|$$

6. Cálculo del MAPE: Otra alternativa que utilizamos para contrastar el error cometido en la predicción es el error porcentual absoluto medio o MAPE (*Mean Absolute Percentage Error*). La fórmula para el cálculo es la siguiente:

$$MAPE\ [\%] = \frac{100}{N}\sum_{k=1}^{N}\left|\frac{y_k - \hat{y}_k}{y_k}\right|$$

   De la misma forma que las anteriores, compara el valor real y el pronosticado y a diferencia de las otras ecuaciones, este indicador mide el tamaño del error en términos porcentuales y es por ello que es altamente utilizado por su fácil interpretación. Con lo cual, esta medida será en gran parte nuestra señal de referencia, sabiendo que, si ha disminuido respecto a otro modelo es porque hemos mejorado en precisión.

Para acabar, se comenta un análisis que ha servido para saber en qué estado se encuentra la red neuronal; si se ha ajustado bien a los datos o no. El uso de redes neuronales tiene un gran potencial, pero es muy complejo dar con la combinación adecuada. Las posibilidades y diferentes combinaciones para crear una red son casi infinitas y por ahora no existen reglas que permitan saber qué es lo más conveniente en cada caso.

Es por ello que la mejor configuración se va obteniendo por prueba y error, hasta llegar al modelo en el que sea más preciso y así, quedarnos con aquellas características que hacen ajustarse mejor a nuestro objetivo.

Esto supone invertir mucho tiempo ya que a medida que aumentamos la complejidad, se incrementa considerablemente el tiempo de computación y potencia necesaria para llevar a cabo el entrenamiento. Para evitar hacer repetidas pruebas, es posible obtener algunas aproximaciones para saber dónde está el límite entre un buen ajuste y un sobreajuste.

Podemos valernos de algunas herramientas analíticas, la más usada consiste en dividir el *data set* principal en tres partes según se ha explicado y dibujar el error de *Train* y de *Validation*. El error de entrenamiento es el error que se obtiene cuando se vuelve a ejecutar el modelo entrenado en los datos de entrenamiento. Como el algoritmo ya ha visto esos datos y mantiene esa experiencia, es de esperar que se tenga un mejor resultado (menor error). De lo contrario, con el error de prueba obtenemos el error que se produce cuando se ejecuta el modelo entrenado sobre un conjunto de datos al que nunca antes estuvo expuesto y por lo tanto es natural que sea un error mayor. En cuanto a los datos de validación son aquellos con los que evaluamos el modelo antes de aplicar el algoritmo sobre los datos de prueba y nos sirve para determinar cuando detener el entrenamiento para evitar la existencia de sobreajuste.

Por lo tanto, podemos decir que, si el modelo trabaja bien con los datos de entrenamiento, pero su precisión es notablemente más baja con los datos de validación (error mucho más grande) se debe a que el modelo ha memorizado los datos que ha visto y no ha podido generalizar para predecir los datos que no ha visto. Y aquí reside la importancia de contar con dos conjuntos de datos distintos antes de aplicar el modelo sobre los datos de prueba; uno para entrenar el modelo y otro para validar su precisión. Con ello entonces, podemos ver cómo se comporta el modelo con datos que nunca ha visto.

A continuación, se muestra un ejemplo en el que se ha sobreentrenado la red y podemos contemplar los diferentes estados:

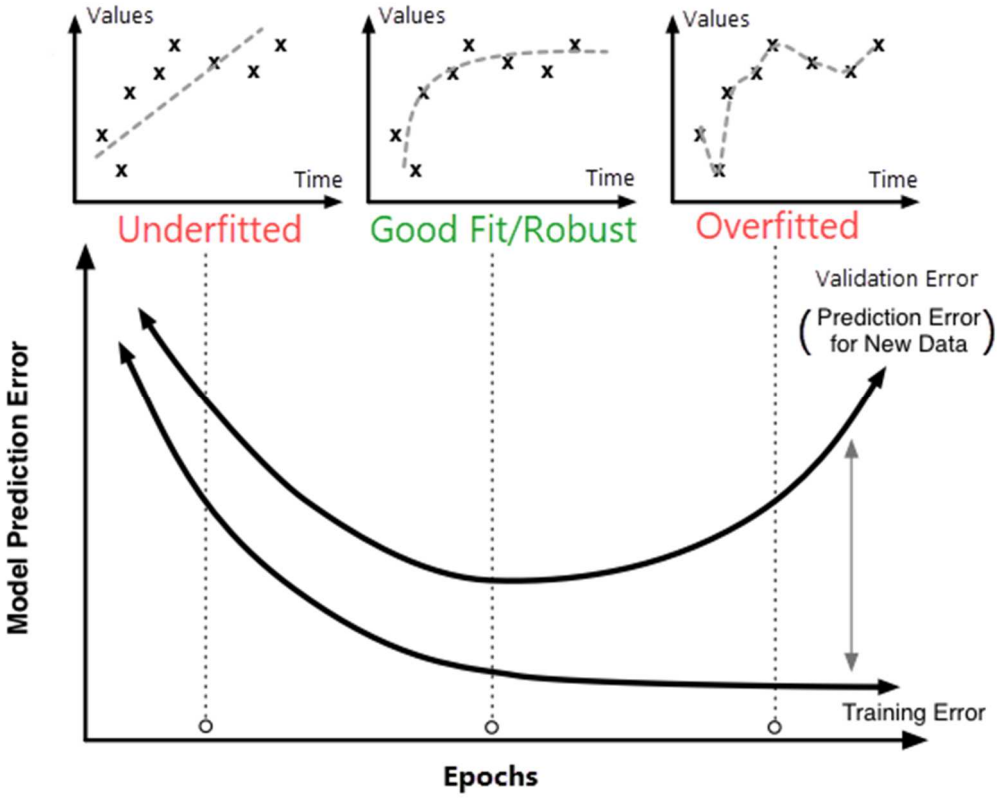


Figura 18: Ejemplo de los posibles ajustes del modelo [13]

En esta figura podemos ver que el error de validación será un poco mayor que el error de entreno, lo cual es evidente (es el resultado de aplicar el algoritmo sobre datos nuevos), pero en el momento que el error de validación empieza a crecer y el margen entre el error de entreno y el de validación aumenta, es porque estamos sobreentrenando la red y es el momento en el que hay que parar de entrenar.

Visto esto, ya tenemos los ingredientes necesarios para empezar con la creación y análisis de los diferentes modelos de *Machine Learning* propuestos.

3.5 Aplicación de un Regresor Lineal

Como primera prueba, utilizamos un algoritmo de regresión ya definido por la biblioteca Scikit-learn. Como ya se ha comentado anteriormente, Scikit-learn es una biblioteca de *Machine Learning* que presenta varios algoritmos de regresión, incluyendo modelos como *Linear Regressor* o *Random Decision Forest* [15]. Empezamos probando nuestros datos con estos tipos de algoritmos por las siguientes razones:

- Se tratan de algoritmos ya creados; no supone prácticamente tiempo de diseño
- Mayor nivel de robustez; no dependen de hiperparámetros
- Buen rendimiento. La matriz con la que trabajamos es de 1389 *rows* x 6 *columns* por lo que el coste computacional es bajo (tenemos un *data set* pequeño)

Con lo cual, estas primeras pruebas nos servirán para ver un antes y un después en nuestros algoritmos; ver qué de buenas son las redes neuronales desarrolladas respecto a un algoritmo ya definido de *Machine Learning*.

Bien, pues el primer modelo con el que trabajamos es con un regresor lineal. La regresión lineal es un modelo atractivo porque la representación es muy simple:

```
regressor = LinearRegression()           # Asignamos a una variable el modelo basado en regresión lineal
regressor.fit(X_train, y_train)          # Entrenamos el modelo con el conjunto de entrenamiento
y_prediction = regressor.predict(X_test) # La predicción es el resultado de aplicar el modelo entrenado sobre los datos de prueba
```

Figura 19: Aplicación de Regresor Lineal

La representación es una ecuación lineal que combina un conjunto específico de valores de entrada (x) cuya solución es la salida pronosticada (y) para ese conjunto de valores de entrada. Como tal, tanto los valores de entrada (x) como el valor de salida (y) son numéricos.

La ecuación lineal asigna un factor de escala a cada columna o valor de entrada, denominado coeficiente y representado por la letra mayúscula griega, Beta (B). También se agrega un coeficiente adicional, dando a la línea un grado adicional de libertad (por ejemplo, moviéndose hacia arriba y hacia abajo en una gráfica bidimensional) y a menudo se denomina intersección o coeficiente de polarización.

Para nuestro caso, al tener 5 entradas, tenemos la siguiente ecuación:

$$y = B_0 + B_1 \cdot x_1 + B_2 \cdot x_2 + B_3 \cdot x_3 + B_4 \cdot x_4 + B_5 \cdot x_5$$

En dimensiones más altas cuando tenemos más de una entrada (x), la línea se llama un plano o un hiperplano. Por lo tanto, la representación es la forma de la ecuación y los valores específicos utilizados para los coeficientes. Ahora que entendemos la representación utilizada para un modelo de regresión lineal, vamos a ver qué tal se ajusta esta representación a partir de los datos.

3.5.1 Resultados y conclusiones

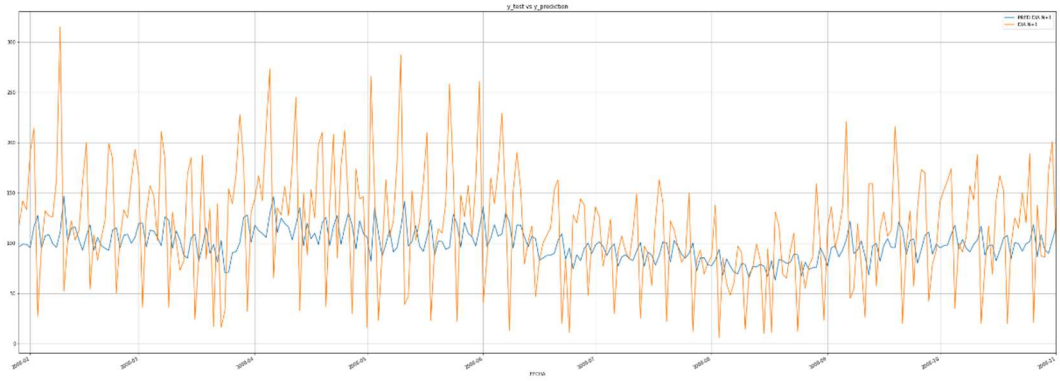


Figura 20: Regresor Lineal - Traza Real vs Predicha

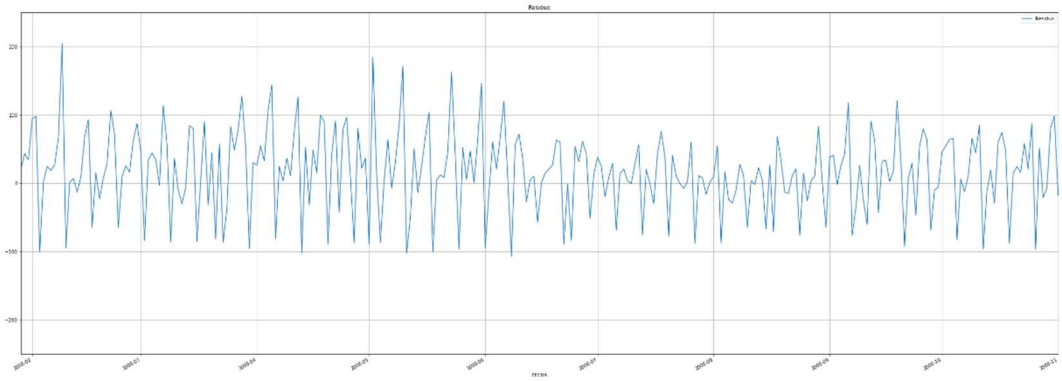


Figura 21: Regresor Lineal - Residuo

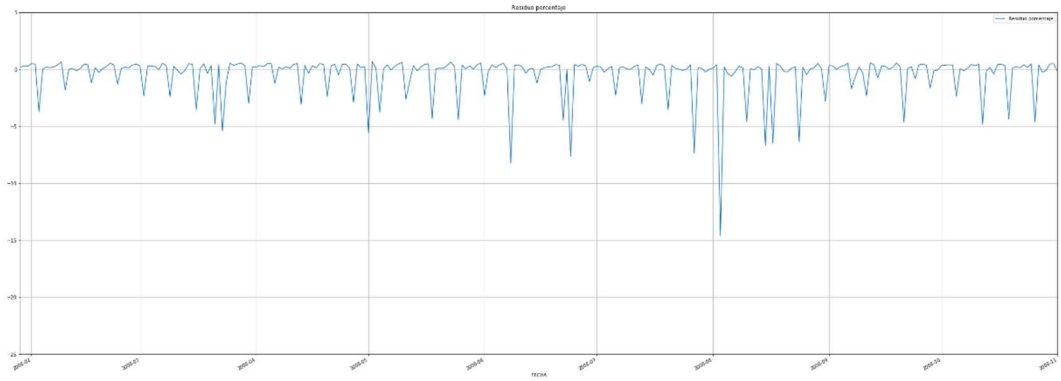


Figura 22: Regresor Lineal - Residuo porcentaje

RMSE	MAE	MAPE
60,80	9,63	17,02%

Figura 23: Regresor Lineal - Resultados

Si miramos la traza de los valores estimados (azul) sobre la traza de los valores reales (naranja), observamos que pese a ser una mala predicción, se sigue una pequeña tendencia. No obstante, no deja de ser una predicción pésima; según la gráfica del residuo, obtenemos prácticamente el mismo dibujo que si graficamos la traza de valores reales, lo cual indica que existe una gran diferencia entre los valores reales y los predichos.

Por otro lado, si analizamos el residuo porcentaje, podemos ver que existen numerosos picos durante todo el tiempo lo cual es otro indicio de que hay muchas discrepancias entre valores estimados y observados.

El mal funcionamiento del modelo se debe en parte a que los datos deben estructurarse de una cierta manera, ya que [16]:

- La regresión lineal **supone que la relación entre su entrada y salida es lineal**. No es compatible con nada más.

- La regresión lineal **asume que sus variables de entrada y salida no son ruidosas**. Se pueden usar operaciones de limpieza de datos que permitan aclarar la señal en nuestros datos. Si es posible, es deseable eliminar valores atípicos.

- La regresión lineal se ajustará a los datos cuando se tengan **variables de entrada altamente correlacionadas**. Es decir, que entre una muestra y otra la diferencia sea la mínima, obteniendo así una traza lo más progresiva posible, evitando cambios bruscos (ruido).

- La regresión lineal hará **predicciones más acertadas si sus variables de entrada y salida tienen una distribución gaussiana**. En otras palabras, que las muestras con mayor peso (en este caso cantidad de yogures) estén próximas entre ellas, sin mezclar o intercalar cantidades muy altas con cantidades muy bajas.

- Y en consecuencia, la regresión lineal a menudo hará **predicciones más precisas si se reescalan las variables de entrada** usando estandarización o normalización.

Es decir, podemos conseguir ciertas mejoras variando el orden de nuestros datos, por ejemplo, de menor a mayor creando una tendencia menos ruidosa y más lineal. Sin embargo, esto no nos interesa ni nos conviene ya que trabajamos con fechas y el orden de estas no se pueden manipular o alterar debido a que afectaría en nuestro resultado final y según este estudio, es el orden establecido (tenemos que trabajar con las mismas condiciones para hacer comparaciones justas).

Por lo tanto, llegamos a la conclusión que no conviene emplear este modelo a nuestro conjunto de datos. Por lo tanto, vamos a probar con otro algoritmo para predicción de *Machine Learning* que pueda entrenar más rápido y mejor, el *Decision Tree Regressor*.

3.6 Aplicación de un Árbol de Decisión

Los árboles de decisión y su extensión (*random decision forests*) pertenecen a la familia de algoritmos de aprendizaje supervisado [17]. Son algoritmos de *Machine Learning* robustos y fáciles de interpretar y a diferencia de otros modelos de aprendizaje supervisado, se utilizan tanto para tareas de clasificación como para tareas de regresión. Los árboles de decisión constituyen una forma simple y rápida de aprender una función, que asigna los datos (x) a las salidas (y), donde (x) puede ser una combinación de variables categóricas y numéricas e (y) puede ser categórico para la clasificación, o numérico para la regresión.

Por ejemplo. Para nuestro caso, cuando el algoritmo reciba el conjunto de datos de entrada acabará aprendiendo que para el domingo es el día con menos demanda, para el martes se compra un poco más y que es el viernes y el sábado el día con mayor demanda. Es decir, cuando reciba un valor muy bajo podrá identificarlo y etiquetarlo como domingo y que al cabo de dos muestras más (martes) la cantidad deberá ser más grande y después de 5 y 6 muestras respecto al domingo (viernes y sábado) todavía un poco más, aproximadamente el doble que la del martes. Y así sucesivamente sabe que cantidad de yogures puede tomar la siguiente muestra a partir de las anteriores recibidas. Y esto lo consigue dividiendo el conjunto de entrenamiento en subconjuntos de tal manera que cada subconjunto contenga datos con el mismo valor (o parecido) para un atributo.

A continuación se muestra un ejemplo de como utiliza los valores de las columnas de entrada para predecir los valores de una columna que se designa como elemento de predicción. Dada una secuencia de entrada con dichos valores, sabrá asignar la salida.

Figura 24: Ejemplo *Decision Tree Regressor*

Es decir, se trata de un modelo predictivo que emplea técnicas de análisis discriminantes dividiendo los datos hasta dar con la solución. Sin embargo, a pesar de su poder contra conjuntos de datos más grandes y más complejos, las redes neuronales pueden tomar muchas iteraciones y ajustes de hiperparámetros antes de obtener un buen resultado (lo veremos más adelante) a diferencia de estos modelos. Una vez sabemos como trabaja el árbol de decisiones, usaremos el paquete Scikit-learn para implementar un árbol de decisión simple y así, predecir el conjunto de datos mostrado anteriormente.

3.6.1 Resultados y conclusiones

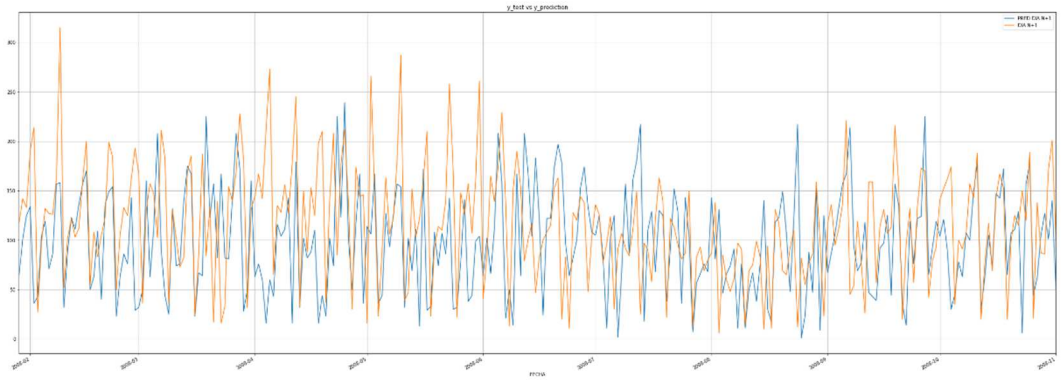


Figura 25: Árbol de Decisión - Traza Real vs Predicha

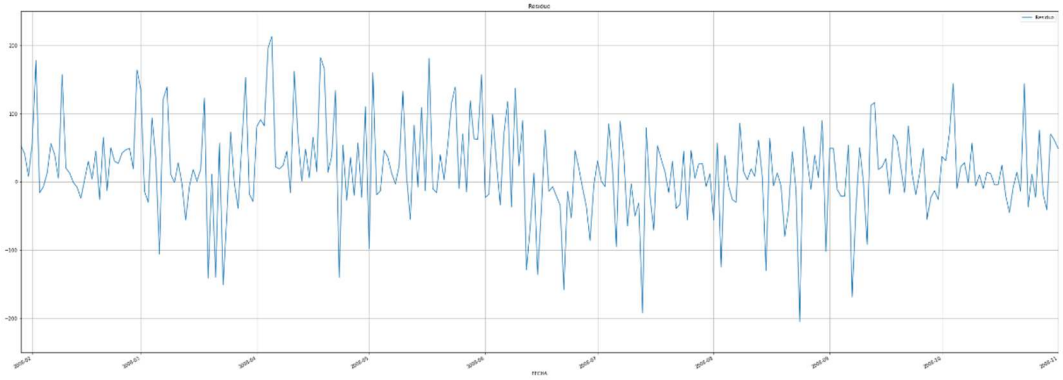


Figura 26: Árbol de Decisión - Residuo

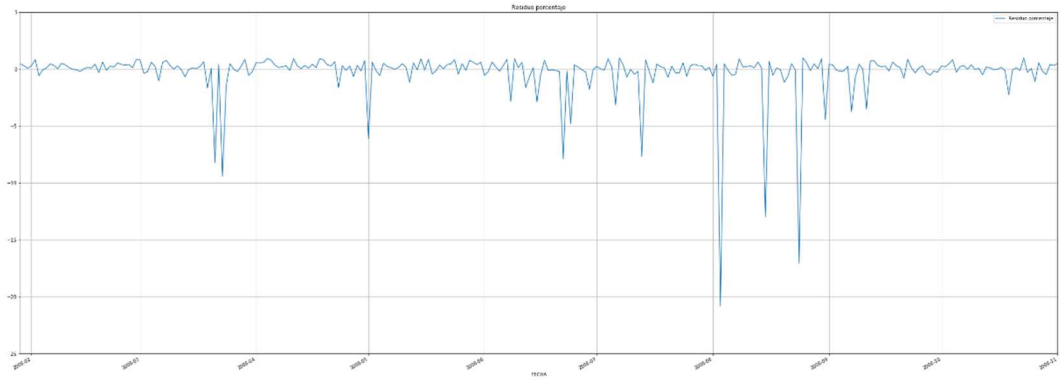


Figura 27: Árbol de Decisión - Residuo Porcentaje

RMSE	MAE	MAPE
69,66	10,16	15,94%

Figura 28: Árbol de Decisión - Resultados

Visualmente, si nos fijamos en el trazado real (naranja) sobre el predicho (azul) (Figura 25), con este algoritmo conseguimos una cierta mejoría respecto al anterior. Sin embargo, es importante analizar también otros parámetros; no nos podemos guiar únicamente por ese gráfico.

Mirando el residuo porcentaje, vemos una cierta mejoría. No obstante, es una medida que puede llevar a confusiones. Si nos paramos a ver su fórmula, nos damos cuenta que para casos en los que el valor predicho es más grande que el valor real, se detecta muy bien a través de los picos. Sin embargo, si es al revés, dichas fechas donde ocurre esto pasan desapercibidas.

E.g.:

Si $y_k < \hat{y}_k$: $y_k = 5,\ \hat{y}_k = 145 \rightarrow Res.Por. = (5-145)/|5| = \mathbf{-28}$

$y_k > \hat{y}_k$: $y_k = 145,\ \hat{y}_k = 5 \rightarrow Res.Por. = (145-5)/|145| \cong \mathbf{1}$

Y esto es lo que sucede en estos dos tramos:

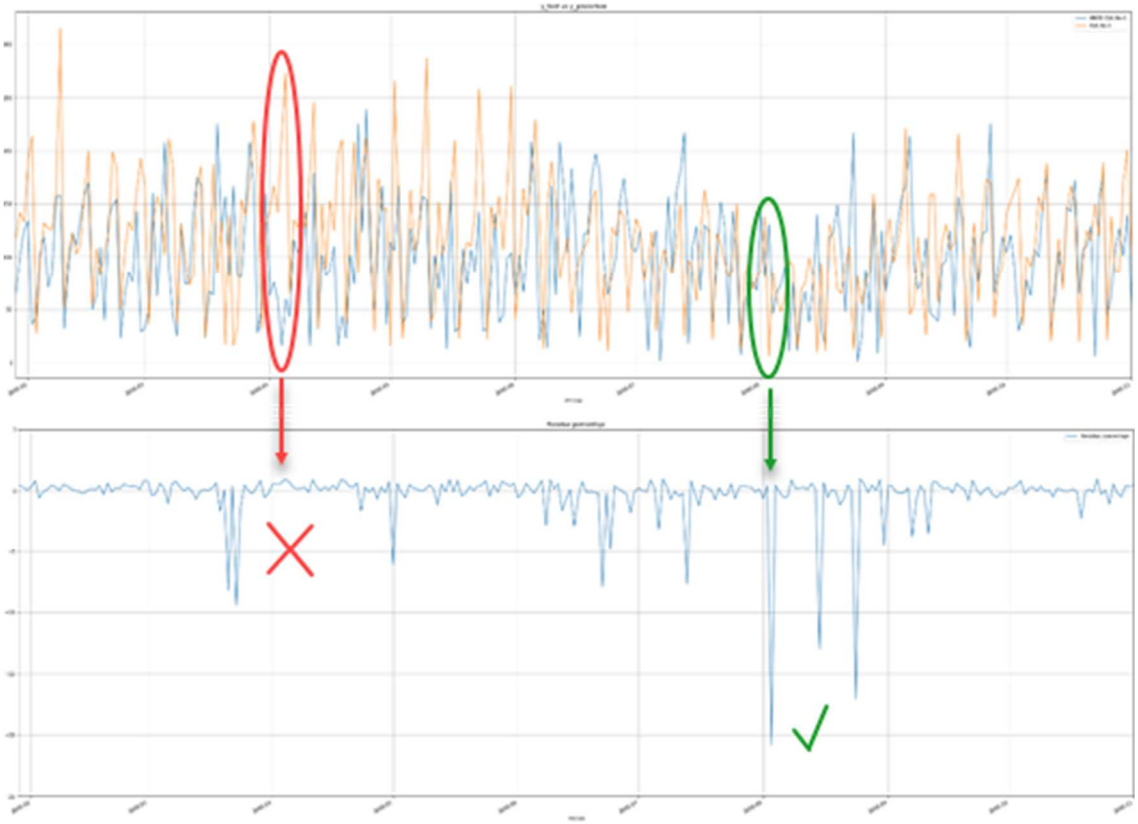


Figura 29: Interpretación residuo porcentaje

Es por ello que el análisis se debe contrastar con diferentes herramientas, y saber cuál va bien y cual va mal para cada caso. Por lo tanto, hay que seguir viendo los resultados obtenidos para valorar el modelo.

Como hemos visto, pese a visualizar un mejor resultado hay que notar que el RMSE (error cuadrático medio) anterior era de 60,80 y ahora es de 69,66. Esta medida nos da una idea de las diferencias entre valores predichos por un modelo y los valores reales. Podemos ver que en la mayoría de instantes se obtiene una mejor predicción respecto al anterior modelo, pero en ciertas tramas, los resultados entre valores reales y predichos son prácticamente opuestos, lo cual hace un empeoramiento del RMSE importante y esto hace que dicha medida se vea perjudicada respecto al anterior algoritmo. Ejemplos de estas tramas son los siguientes:

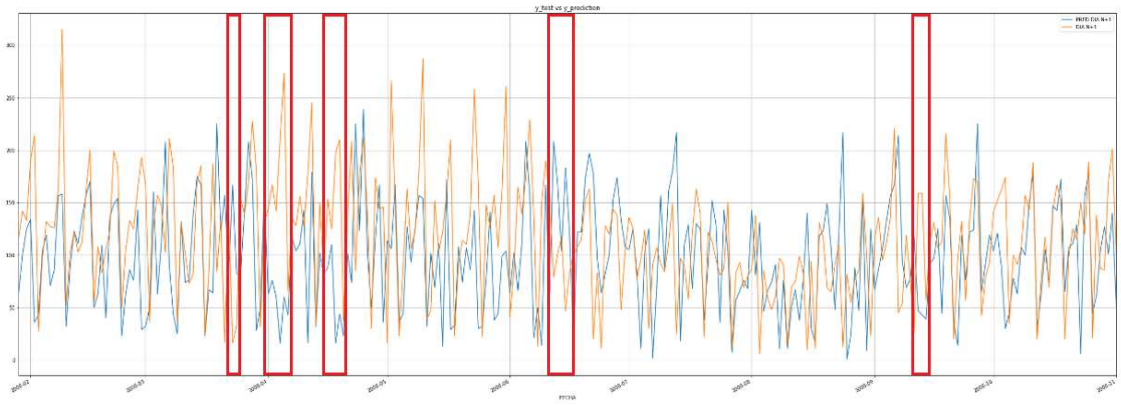


Figura 30: Tramas donde los resultados son opuestos

En cuanto al MAE (error absoluto medio), el hecho de tener mayor variación en la predicción hace que esta medida también se vea incrementada (de 9,63 a 10,16) ya que nos mide la diferencia entre dos variables continuas y en este caso, tenemos una señal predicha más ruidosa que la anterior.

Tomar la raíz cuadrada del promedio de los errores cuadrados tiene algunas implicaciones interesantes para RMSE. Dado que los errores se elevan al cuadrado antes de promediarlos, el RMSE otorga un peso relativamente alto a los grandes errores. Esto significa que el RMSE debería ser más útil cuando los grandes errores son particularmente indeseables.

Por último, examinemos el MAPE (error porcentual absoluto medio). Esta medida será la que nos ayude a valorar a groso modo si hemos mejorado o no, pues mide la precisión de un método de pronóstico. En este caso respecto al anterior, vemos que existe una pequeña mejoría de 17,02% a 15,94% que es lo que se esperaba viendo la Figura 25 respecto a la 20. Aun así, pese a haber mejorado con la precisión, no nos conformamos con el resultado, por lo que decidimos implementar una red neuronal para conseguir una mejor predicción.

Usar redes neuronales implica mayor dificultad a la hora de diseñar nuestro algoritmo, no obstante también permite mayor flexibilidad. A continuación, veremos si es beneficioso utilizar dichas arquitecturas para mejorar en precisión. Para ello empezaremos con un tipo de red simple, el perceptrón multicapa.

3.7 Aplicación de un Perceptrón Multicapa

Un MLP es una red de neuronas simples llamadas perceptrones. El perceptrón calcula una única salida de múltiples entradas de valores reales formando una combinación lineal de acuerdo con sus pesos de entrada y colocando a la salida alguna función de activación no lineal según se ha explicado en la sección 2.1.1 (Figura 3). Una red típica de perceptrón multicapa (MLP) consiste en un conjunto de nodos fuente que forman la capa de entrada, una o más capas ocultas de nodos de computación y una capa de salida de nodos. La señal de entrada se propaga a través de la red capa por capa dando como resultado un valor en la salida. Para nuestro caso en concreto nos encontraremos con la siguiente estructura:

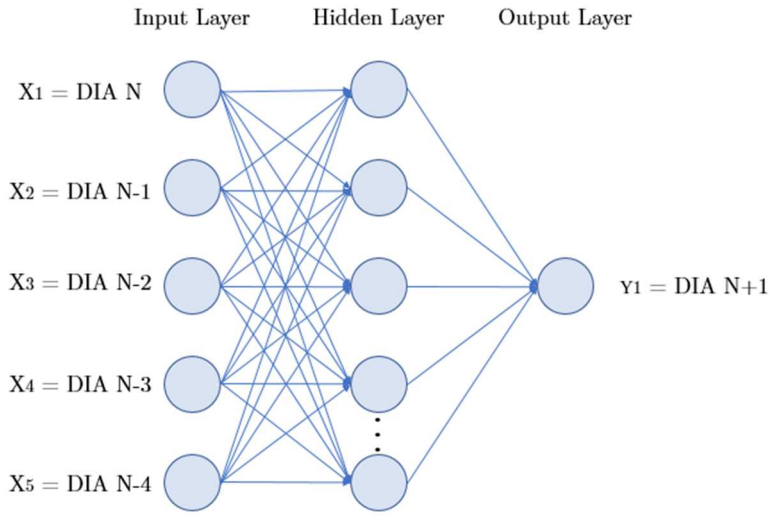


Figura 31: Arquitectura MLP

Este tipo de redes utiliza la propagación hacia atrás del error o retropropagación (*backpropagation*) que como se comentó, la señal de salida se compara con la salida deseada y se calcula el error, propagándolo hacia atrás capa por capa para rectificar los pesos de forma repetida en el proceso de entreno.

El mayor reto después de diseñar la estructura de nuestra red neuronal es optimizarla. Nos encontramos con que hay muchos parámetros de la red para sintonizar. En redes neuronales se conocen como hiperparámetros y se puede entender que son los controles que los programadores ajustan en los algoritmos de aprendizaje automático para afinar la red (algunos ejemplos son el número de capas ocultas en la red, el número de neuronas en cada capa, el *learning rate*, etc). La mayoría de ellos se van a sintonizar a medida que vayamos realizando pruebas, pues cada *data set* tiene un comportamiento diferente y es difícil aplicar un criterio para elegir estos valores. Por lo tanto, se trata de un proceso de prueba y error, pero debemos de saber o tener una idea de qué parámetros hay que tocar para ver cambios relevantes en nuestros resultados. También hay que tener en cuenta que no hay un valor óptimo o mejor para un parámetro; como se ha dicho anteriormente,

cada red neuronal es distinta. Un buen comienzo es modificar el número de épocas. Recordar que una época es un paso único en el entrenamiento de una red neuronal; en otras palabras, no es más que una iteración de un paso para todo su conjunto de datos de entrada. Entonces el proceso de entrenamiento puede consistir en más de una época. Dar el número adecuado de épocas es importante ya que un número bajo implica una red neuronal poco entrenada y por consecuencia unos peores resultados. Por el contrario, un número alto supone un coste computacional mayor y quizás un sobreajuste en nuestro algoritmo. Con lo cual, debemos encontrar un equilibrio a la hora de entrenar nuestros datos. Para nuestro estudio se ha empleado dos casos extremos un entreno con 10 épocas y otro entreno con 1000 épocas para corroborar esta teoría, no obstante como se introdujo, el número de épocas correcto se puede determinar a partir de la curva de error de entreno y la de validación.

Otro parámetro importante es la tasa de aprendizaje (*learning rate*). Dicho parámetro, indica cómo de rápido aprende el algoritmo. Un valor alto no implica un mejor resultado, puede dar a un algoritmo más "nervioso". Es decir, se conoce como la rapidez con que una red abandona las viejas creencias por nuevas; una tasa de aprendizaje más alta significa que la red cambia de opinión más rápidamente. Otra vez, hay que encontrar un *learning rate* lo suficientemente bajo como para que la red converja algo útil, pero lo suficientemente alto como para que no tenga que pasar años entrenando. Los valores típicos pueden variar del orden de 0,0001 hasta 1. En nuestro caso hemos usado 2 valores distintos que permiten ver esa explicación teórica en nuestros resultados; la diferencia de usar un *learning rate* de 0,001 a uno de 0,0001.

Por último, otro parámetro de interés es el número de capas ocultas y el número de neuronas de estas. Un incremento de profundidad de la red (mayor número de *hidden layers*) no necesariamente implica un mejor resultado. Agregar capas aumenta el número de pesos en la red y en consecuencia una red cada vez más grande, con lo que puede llevar a un sobreajuste y empeorar la precisión. En nuestro caso en particular y en general para data sets no muy complejos ni grandes, una capa oculta es suficiente para la mayoría de problemas. Para nuestros datos está comprobado que no existe margen de mejora variando el número de capas ocultas, por lo que decidimos variar la cantidad de neuronas de nuestra capa. Estableciendo una única capa oculta, como hemos dicho, se puede modificar el número de neuronas de ésta. Dos valores que nos permiten ver distintos resultados en nuestros datos son de 50 y de 200 *hidden neurons*. Aclarar que se han hecho muchas más combinaciones de todos estos parámetros para probar resultados, pero se comenta lo más interesante; mencionarlas todas sería excesivo. Y según se ha dicho al principio, estos parámetros han sido variados con un cierto criterio dentro de los posibles parámetros a modificar, tomando estos valores como los interesantes después de ciertas pruebas. Después de haber empleado diferentes configuraciones para ver que combinación es la óptima, para MLP el mejor resultado obtenido ha sido el siguiente.

3.7.1 Resultados y conclusiones

La configuración usada ha sido 200 *epochs*, 50 *hidden neurons*, *learning rate* de 0,001.

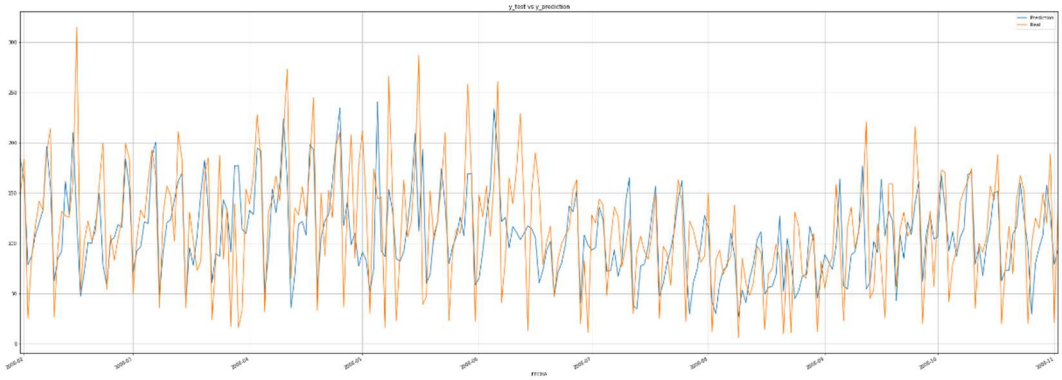


Figura 32: MLP - Traza Real vs Predicha

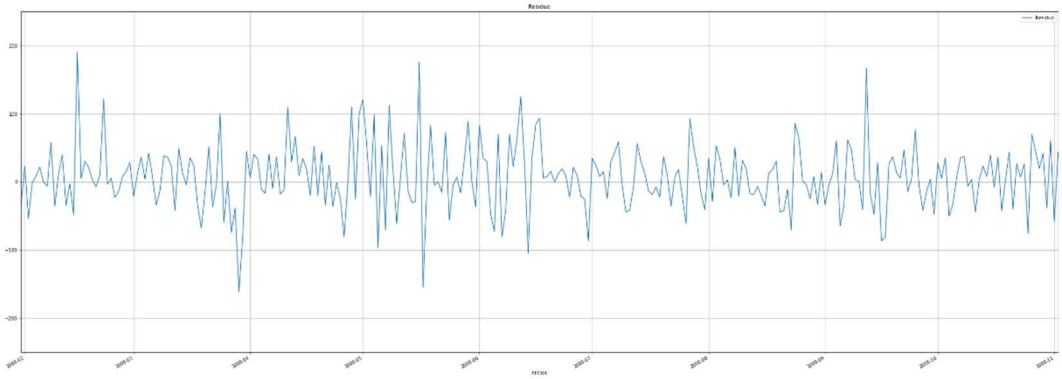


Figura 33: MLP - Residuo

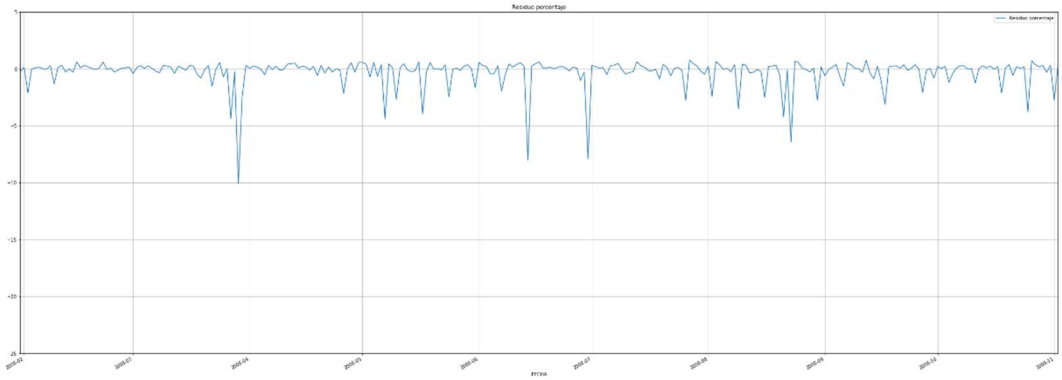


Figura 34: MLP – Residuo Porcentaje

RMSE	MAE	MAPE
40,20	6,17	10,03%

Figura 35: MLP – Resultados

Vamos a analizar estos resultados:

- Aparentemente la predicción se adapta más a lo esperado respecto al anterior modelo, tenemos un punto a favor.
- La traza del residuo presenta una forma más plana y regular.
- Reducción importante del RMSE. Pasamos de 69,66 a 40,20, por lo que tenemos menor diferencia entre el valor estimado y el real. Otro punto a favor.
- Un mejor MAE. De 10,16 a 6,17. Nos indica que hay menos diferencia entre variables continuas, tenemos una predicción más "controlada" y menos "nerviosa" evitando los cambios bruscos que veíamos antes.
- Una reducción del MAPE de 15,94% a 10,03%. Es una mejora notable, hemos mejorado la precisión de nuestro algoritmo en 6 puntos.

Una explicación teórica que corroboramos después de estos resultados es que ha sido mejor emplear 50 *hidden neurons* en lugar de 200. Como hemos dicho anteriormente, un mayor número de neuronas o de capas no necesariamente da un mejor resultado.

Otro punto interesante a mencionar es el número de épocas usado. Para empezar, se probó hasta 1000 épocas de entreno como se ha dicho y pintando el error de entreno frente al de validación se corrigió a 200 épocas. Este fue el resultado:

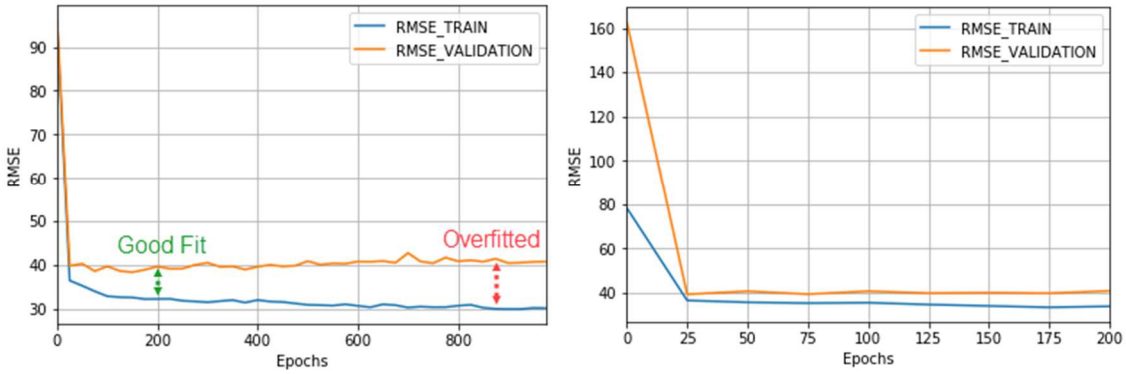


Figura 36: 1000 epochs (Overfitting) vs 200 epochs (Good fit)

Podemos ver como en la figura izquierda en la época 200, el error de validación empieza a crecer poco a poco y el error de entreno a disminuir. A medida que avanzan las épocas el margen entre estos errores se hace más grande indicando que el algoritmo se ha sobreentrenado ya que cuando recibe unos datos que ya ha visto el error de predicción es mucho más bajo que cuando predice sobre unos datos que nunca ha visto y por lo tanto se ha acostumbrado a los datos de entreno y no generaliza para nuevas entradas.

Como conclusión, después de ver este modelo podemos decir que ha sido un éxito emplear redes neuronales. Hemos visto una mejora en todas las medidas y resultados. Aun así, parece que existe un margen de mejora. Esto puede hacerse empleando otro tipo de red neuronal, las redes neuronales recurrentes.

3.8 Aplicación de una Red Neuronal Recurrente

Las redes neuronales recurrentes (RNN) tienen la peculiaridad de que tienen caminos de retroalimentación entre todos los elementos que las conforman. Es decir, la señal de salida para una neurona se realimenta hacia la entrada de la misma u otras neuronas del mismo nivel [18]:

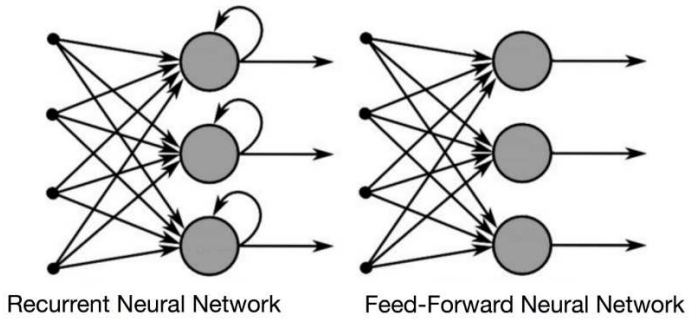


Figura 37: Esquema de una RNN vs *Feed-Forward* NN

Cada neurona está entonces conectada a las neuronas posteriores en la siguiente capa, las neuronas pasadas de la capa anterior y a ella misma a través de vectores de pesos variables que sufren alteraciones en cada *epoch* con el fin de alcanzar los parámetros o metas de operación. Además, es uno de los algoritmos más potentes en el momento porque tienen memoria interna, cosa que permite recordar la entrada que recibieron y esto les ayuda a ser muy precisos en la predicción de lo que vendrá después. A diferencia de las redes neuronales *feedforward*, que no tienen memoria de la entrada que recibieron anteriormente y no pueden recordar nada excepto su entrenamiento.

Esta es la principal razón por la cual las RNN's se están volviendo muy populares. Con estas características, tienen una amplia utilidad para múltiples aplicaciones. Pueden analizar datos de series de tiempo, como precios de acciones y proporcionar pronósticos. En los sistemas de conducción autónomos, pueden anticipar las trayectorias del automóvil y ayudar a evitar accidentes. Pueden tomar frases, documentos o muestras de audio como entrada, lo que los hace extremadamente útiles para los sistemas de procesamiento de lenguaje natural (NLP), como la traducción automática. Si miramos más de cerca lo que sucede, seguidamente se muestra el despliegue de una neurona RNN:

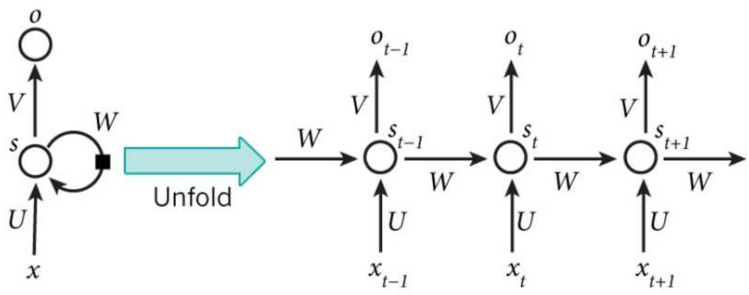


Figura 38: Despliegue de una neurona RNN [18]

Vemos que x_t es la entrada en el instante de tiempo t. Por ejemplo, $x_1 = DIA\ N$ podría ser la primera cantidad de yogures en el período de tiempo 1. s_t es el estado en ese momento y se calcula en función del estado oculto anterior (información del elemento anterior) y la entrada en el paso actual, utilizando una función de activación. s_{t-1} generalmente se inicializa a cero y o_t es la salida en el paso t.

En resumen, cada célula RNN se desarrolla con la idea de que una entrada depende de la entrada anterior al tener un estado oculto, o memoria, que captura lo que se ha visto hasta ahora. Por lo tanto, una RNN tiene dos entradas; el presente y el pasado reciente. Un buen ejemplo para concluir esta pequeña introducción teórica sobre las RNN es el siguiente [18]: Si una red neuronal *feedforward* recibe como entrada la palabra "neurona" y tiene que procesarla carácter por carácter, cuando esté por ejemplo por la letra "r", ya se habrá olvidado de "n", "e" y "u", lo que hace realmente complicado predecir qué carácter vendrá después. Sin embargo, si es una RNN la que recibe dicha palabra y se encuentra por el carácter "r" sabrá que anteriormente recibió una "u" por lo que después del conjunto de caracteres "ur" solo puede venir una "o".

Al igual que en MLP, el modelo consiste en tres bloques principales. La capa de entrada, la capa oculta y la capa de salida. La diferencia entre ambos modelos de redes neuronales se encuentra en la forma que trabaja la arquitectura de red, como hemos dicho, las neuronas de las RNN también permiten que los datos fluyan hacia atrás en la red.

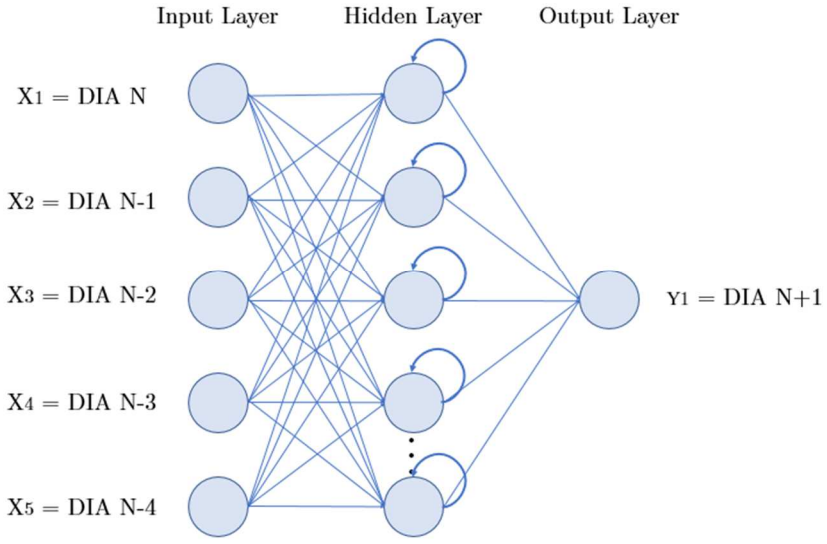


Figura 39: Arquitectura RNN

Y es por ello que, para un mismo conjunto de datos de entrenamiento, entregarán mejores resultados que cualquiera de los otros tipos de algoritmos vistos anteriormente. Para comparar modelos de redes neuronales de forma justa, hemos empleado las mismas combinaciones y en este caso el resultado ha sido el siguiente.

3.8.1 Resultados y conclusiones

La configuración usada ha sido 400 *epochs*, 200 *hidden neurons*, *learning rate* de 0,0001.

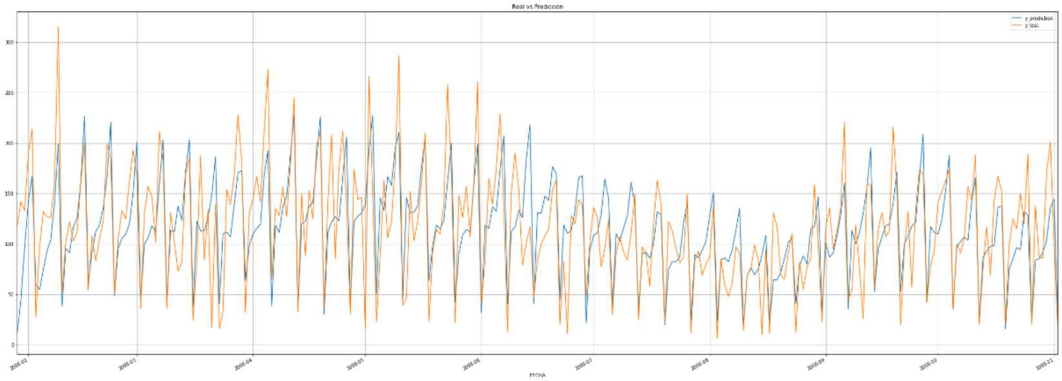


Figura 40: RNN - Traza Real vs Predicha

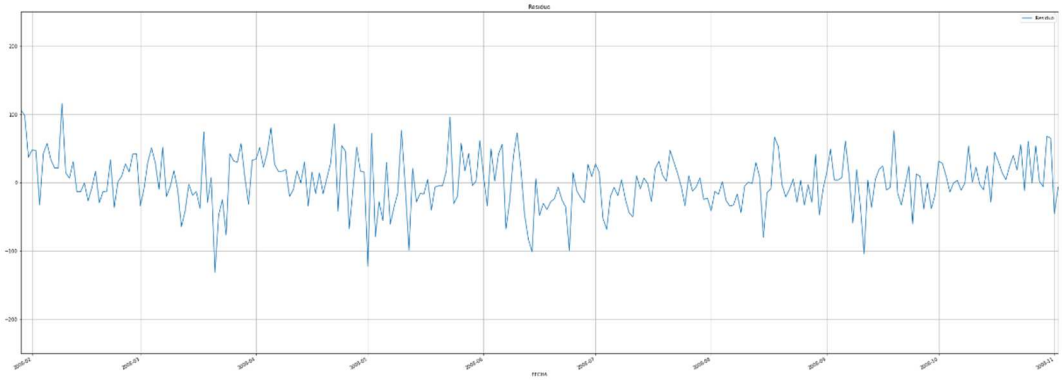


Figura 41: RNN - Residuo

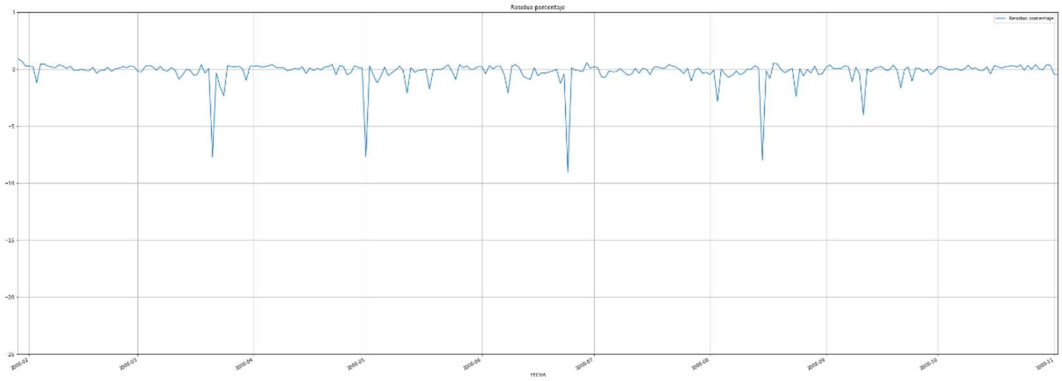


Figura 42: RNN - Residuo Porcentaje

RMSE	MAE	MAPE
38,86	5,95	8,81%

Figura 43: RNN – Resultados

Anteriormente la configuración que proporcionaba un mejor resultado era de 200 *epochs*, 50 *hidden neurons* y un *learning rate* de 0,001, en cambio ahora ha sido de 400 *epochs*, 200 *hidden neurons* y *learning rate* de 0,0001. En el número de épocas hemos visto un mejor resultado incrementándolo, y esto se ha hecho de la misma forma que antes; se empezó probando con 1000 épocas y se ajustó después del análisis de la Figura 44.

Si miramos el número de neuronas vemos que en este caso si ha sido mejor usar una mayor cantidad, pero recordar que, una red más grande y con más neuronas no necesariamente da mejores resultados. Tener más neuronas también puede llevar a un sobreajuste (*overfitting*) en el cual el modelo se ajusta muy bien a los datos existentes y tiene un pobre rendimiento para predecir nuevos resultados. Cosa que no nos conviene para nada ya que, nuestros dados son muy variados y el algoritmo no debe "confiarse" (debe estar preparado a cambios). Respecto al *learning rate*, usar el mismo de antes (0,001) no nos da un mal resultado, pero como buscamos el óptimo, se ha visto que con un *learning rate* de 0,0001 es mejor, con lo cual, nos quedamos con ese.

Si pasemos a analizar las medidas de RMSE, MAE y MAPE respecto a los anteriores vemos que la mejoría es clara. Se ha conseguido reducir el RMSE todavía un poco más lo cual quiere decir que los valores predichos se ajustan más a los reales. El MAE es el justo para nuestros datos, como hemos visto en el anterior modelo se conseguía eliminar los cambios bruscos en la predicción con lo cual el margen de mejora aquí era pequeño. Aun así, se ha conseguido reducir. El MAPE en el anterior modelo ya había dado un buen salto a mejor pero aquí vemos que todavía más. En este caso vemos una predicción más "inteligente".

Y ya para acabar, vamos a ver cómo ha sido el estudio del momento en el que debemos dejar de entrenar, es decir el resultado que nos ha ayudado a escoger el número de épocas de entreno. En este caso se ve más claro lo comentado en la Figura 18:

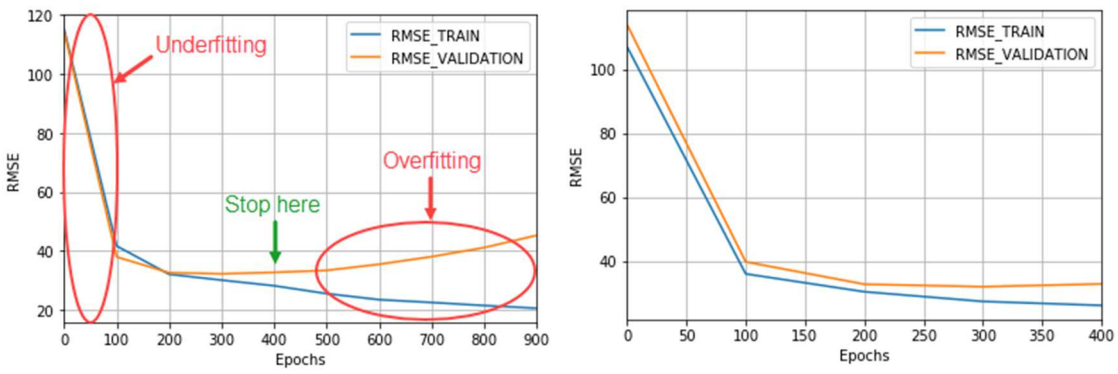


Figura 44: Evaluación a partir del error de entreno y validación

3.9 Aplicación de una Red Neuronal Recurrente Combinada

Como hemos podido ver a lo largo de este trabajo, tener más cantidad de datos puede favorecer el resultado siempre y cuando se puedan procesar. Como último experimento, se trata de observar si es posible aprovechar datos de otras provincias para mejorar la predicción.

Antes de lanzarnos a la piscina, hay que ver si los datos entre provincias están más o menos correlados, si no fuera el caso, no nos serviría de nada tener más datos. Por ello buscamos la provincia con una actividad parecida a la de Barcelona y la que mejor se adapta al experimento es Madrid. Así que construimos un *data frame* para Madrid igual que para Barcelona y calculamos la correlación entre esos datos. Este es el resultado obtenido (Código de provincia: 28 = Madrid, 8 = Barcelona):

	CANTIDAD_Prov 28	CANTIDAD_Prov 8
CANTIDAD_Prov 28	1.000000	0.776036
CANTIDAD_Prov 8	0.776036	1.000000

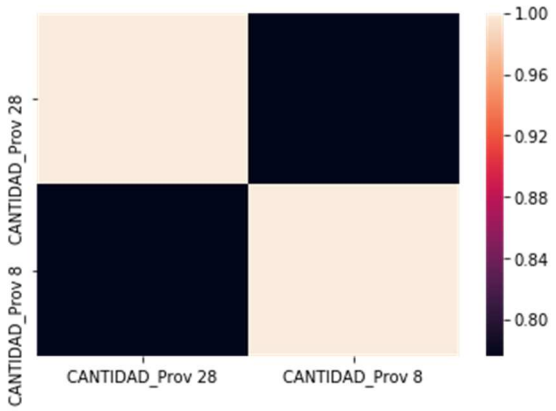


Figura 45: Correlación Madrid - Barcelona

Podemos decir que estas provincias están bien correladas en cuanto a estos datos se refiere ya que un valor de 0,77 indica que comparten bastante similitud. El siguiente paso entonces es crear la red neuronal para ver si esto funciona.

Como se ha visto que la que ha dado un mejor resultado ha sido la RNN, se sigue manteniendo ese modelo, pero en este caso será necesario añadir otras 5 entradas (5 para Madrid y 5 para Barcelona) y otra salida. Es decir, el diseño de la red neuronal combinada será el siguiente:

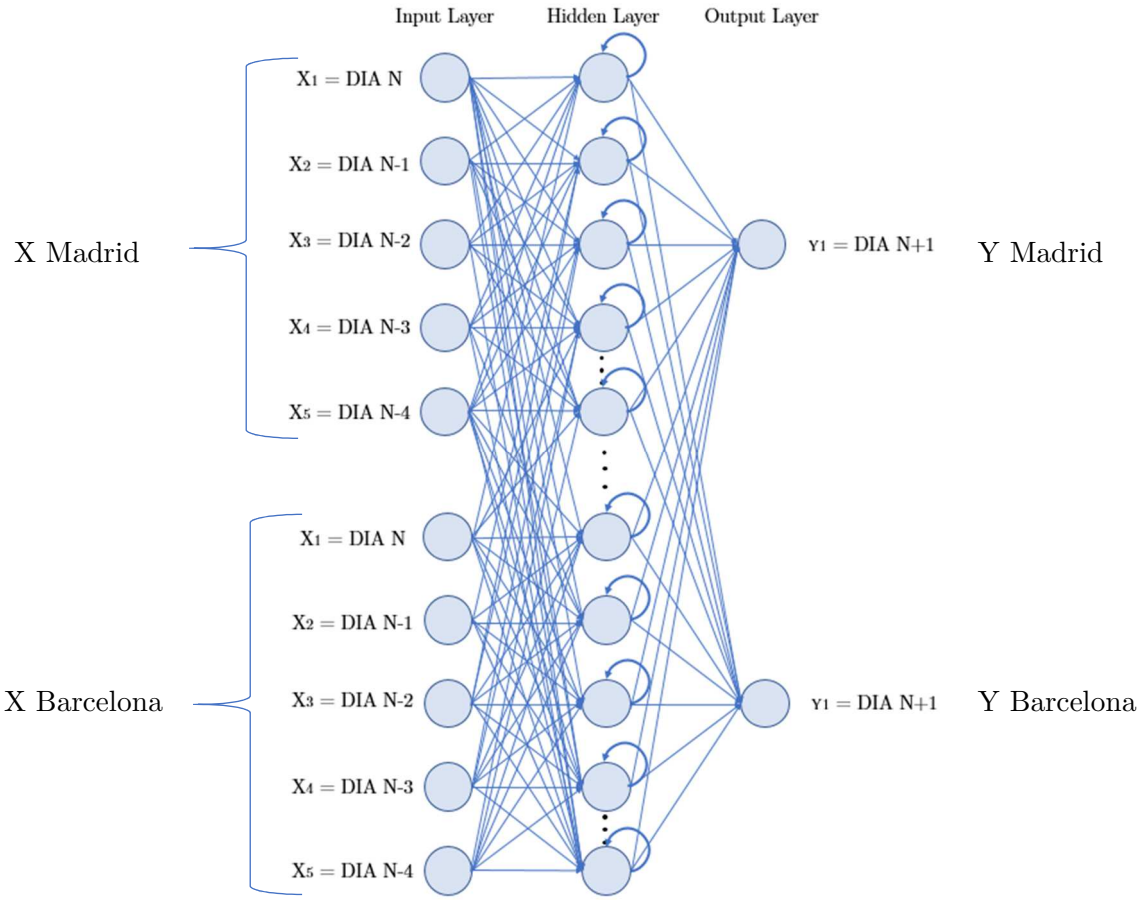


Figura 46: Arquitectura RNN Combinada

Demostramos que es una buena correlación si pintamos las cantidades observadas (reales) del último 20% del *data set*, *test set* (280 últimos días/cantidades de yogures) para Madrid (verde) y Barcelona (naranja); vemos que son muy parecidas, lo cual indica que el hecho de tener los datos de Madrid, puede reforzar a la red neuronal con más ejemplos.

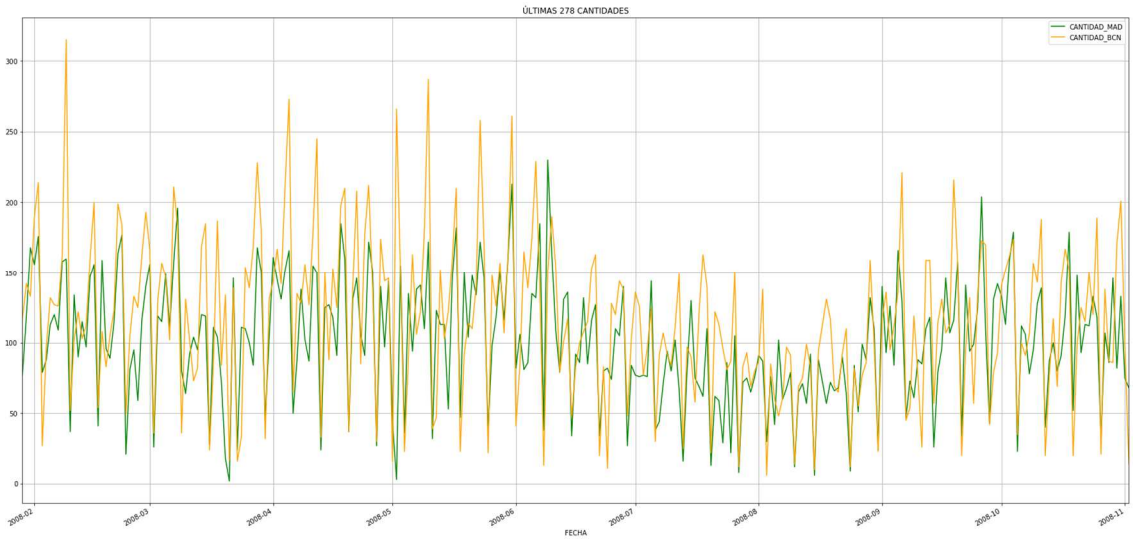


Figura 47: y real de Madrid e y real de Barcelona

3.9.1 Resultados y conclusiones

La configuración usada ha sido 700 *epochs*, 200 *hidden neurons*, *learning rate* de 0,0001.

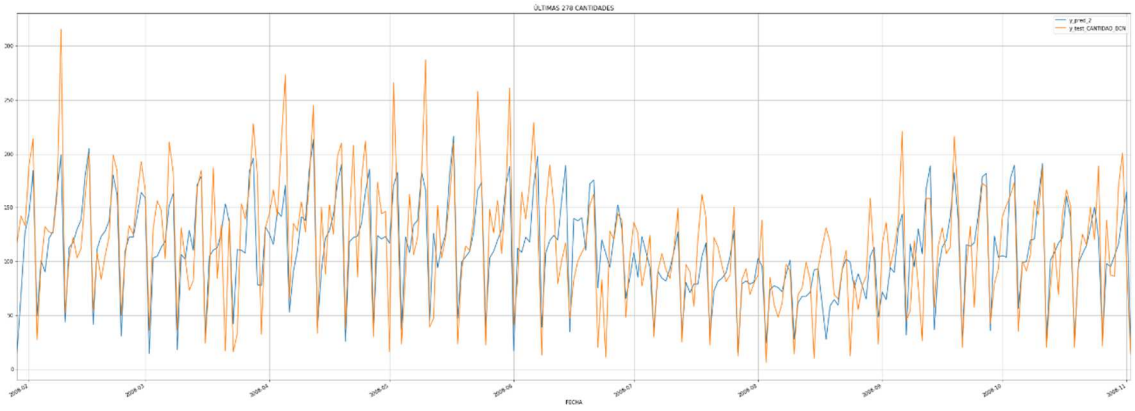


Figura 48: RNN Combinada (BCN) - Traza Real vs Predicha

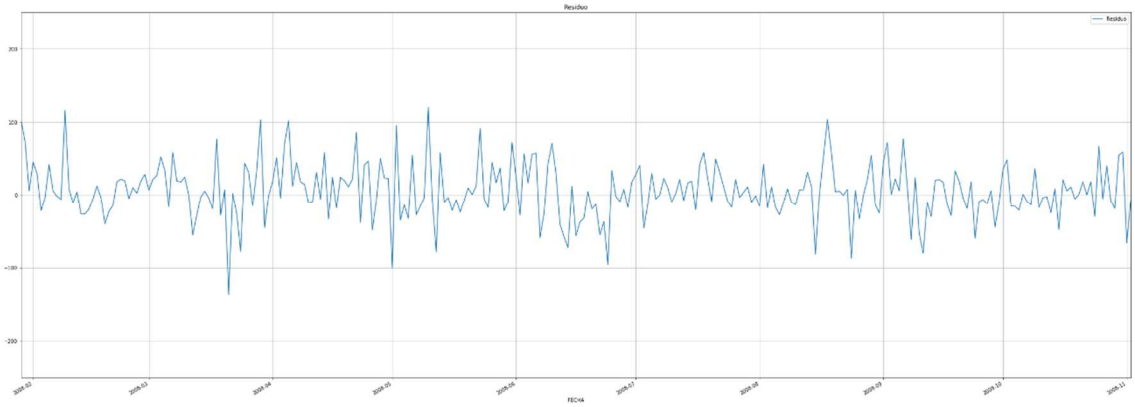


Figura 49: RNN Combinada (BCN) - Residuo

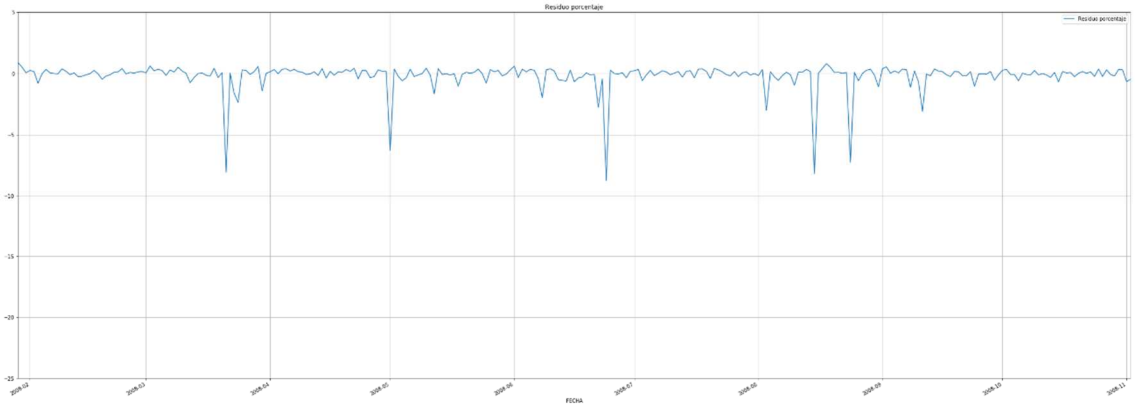


Figura 50: RNN Combinada (BCN) - Residuo Porcentaje

RMSE	MAE	MAPE
38,11	5,63	8,67%

Figura 51: RNN Combinada (BCN) - Resultados

Finalmente nos queda valorar cuan bueno es el modelo. Mantenemos la misma configuración de la red neuronal recurrente anterior que nos ha funcionado bien de 200 *hidden neurons*, *learning rate* de 0,0001. El número de épocas de entreno es el que es distinto; en este caso según el error de validación y entreno, vemos que a partir de las 700 épocas el error de validación empieza a aumentar ligeramente distanciándose del error de validación por lo que decidimos cortar el entreno en 700 épocas.

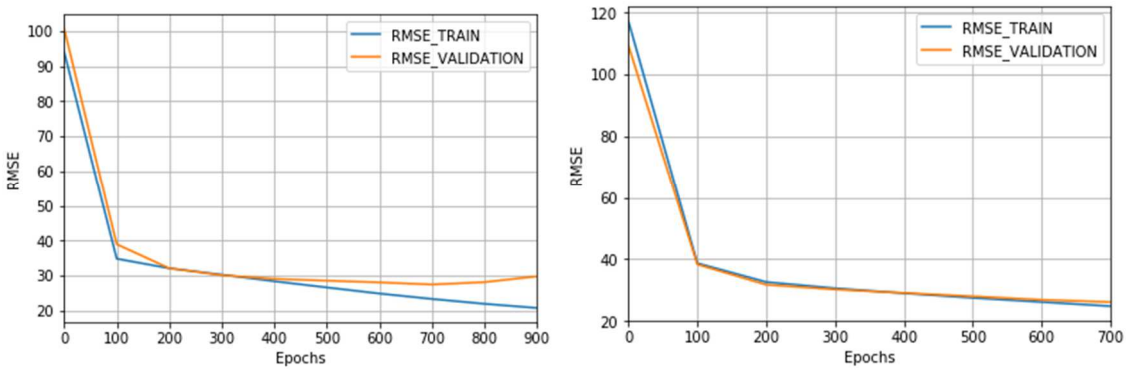


Figura 52: Análisis de ajuste a partir del *train error* y el *validation error*

Como vemos en los distintos resultados obtenidos, hemos mejorado un poco más añadiendo nuevas entradas. Se puede ver una pequeña mejora en todas las medidas, sobre todo en el error de validación, que prácticamente es el mismo que el error de entreno lo cuál indica que la red está generalizando bien.

La red ha sido capaz de entrenar con datos de Madrid y Barcelona, aportando a la predicción una mayor precisión. Hay que tener en cuenta que esto ha resultado favorable dado a la alta correlación entre estas dos provincias; con cualquier otra provincia no era viable por la diferencia que había entre cantidades de yogures. También se hace notar que cada vez cuesta más mejorar el resultado ya que una vez optimizados los hiperparámetros la solución ha sido emplear más datos.

A continuación, se recogen los diferentes resultados para tener una visión más general:

	RMSE	MAE	MAPE
Linear Regressor	60,80	9,63	17,02%
Decision Tree Regressor	69,66	10,16	15,94%
Multilayer Perceptron	40,20	6,17	10,03%
Recurrent Neural Network	38,86	5,95	8,81%
Combined Recurrent Neural Network	**38,11**	**5,63**	**8,67%**

Figura 53: Resultados de los diferentes modelos predictivos

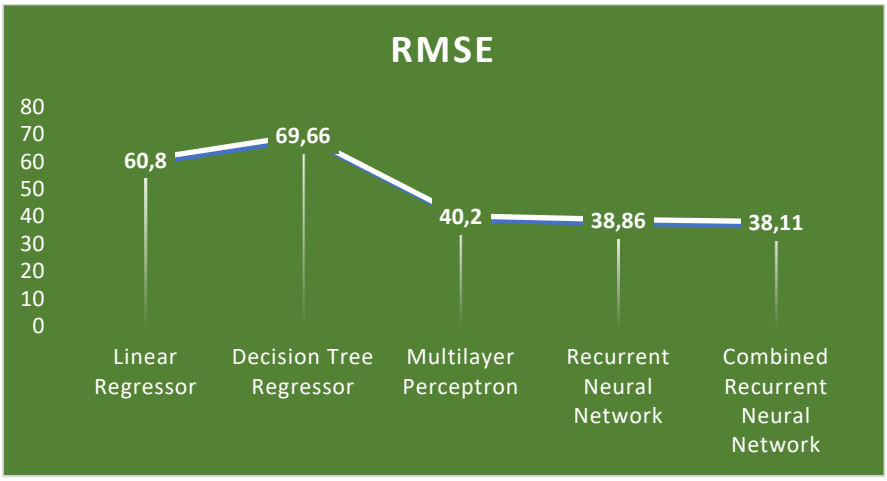


Figura 54: Evolución del RMSE

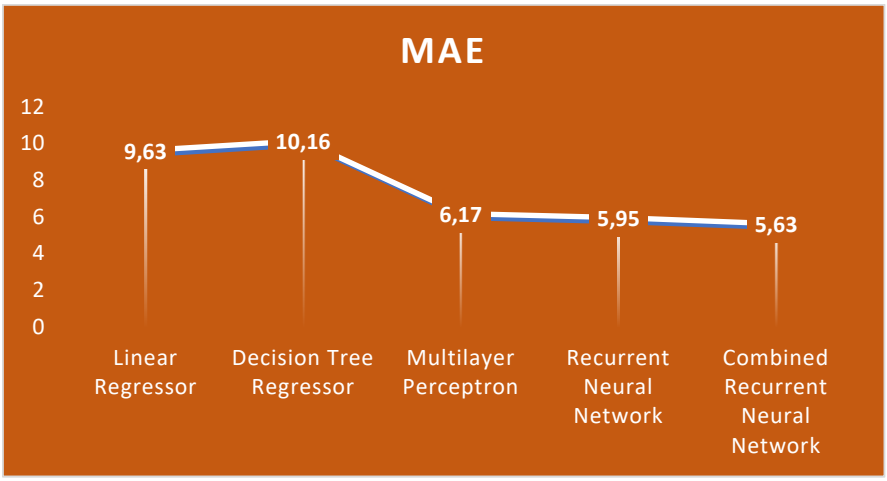


Figura 55: Evolución del MAE

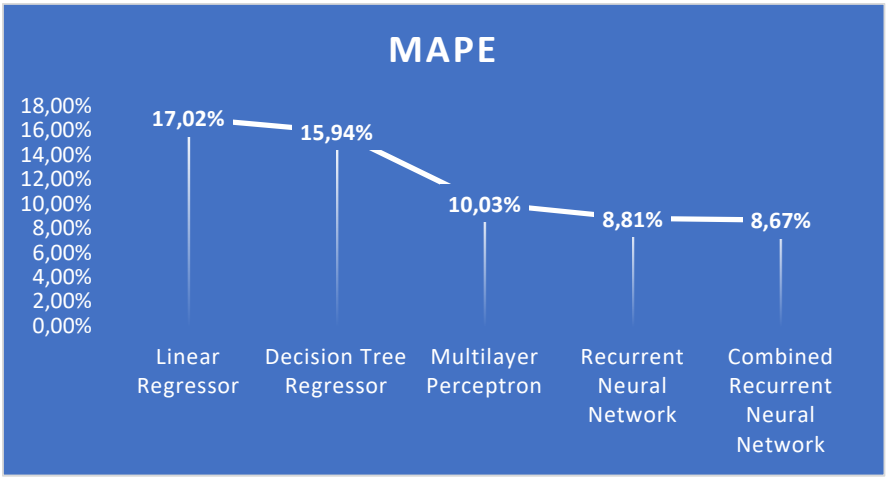


Figura 56: Evolución del MAPE

Después de haber experimentado y trabajado con los distintos modelos, nos damos cuenta que llegamos a un momento en el que pulir pequeñas mejoras de nuestro resultado se hace realmente un reto. Probablemente mejoraríamos bastante si en lugar de tener la recopilación de cantidades de yogures de $\cong$ 4 años fuera de más, y evidentemente la capacidad de computación para procesar dichos datos. En ese caso existe un compromiso entre el coste computacional y la precisión de los resultados. Es decir, para una mayor precisión necesitaríamos más cantidad de datos que esto conllevaría a un mayor tiempo de computación (o a necesitar mejores procesadores).

Por ahora, este recorrido por los diferentes modelos con nuestros datos, deja la siguiente experiencia y reflexión:

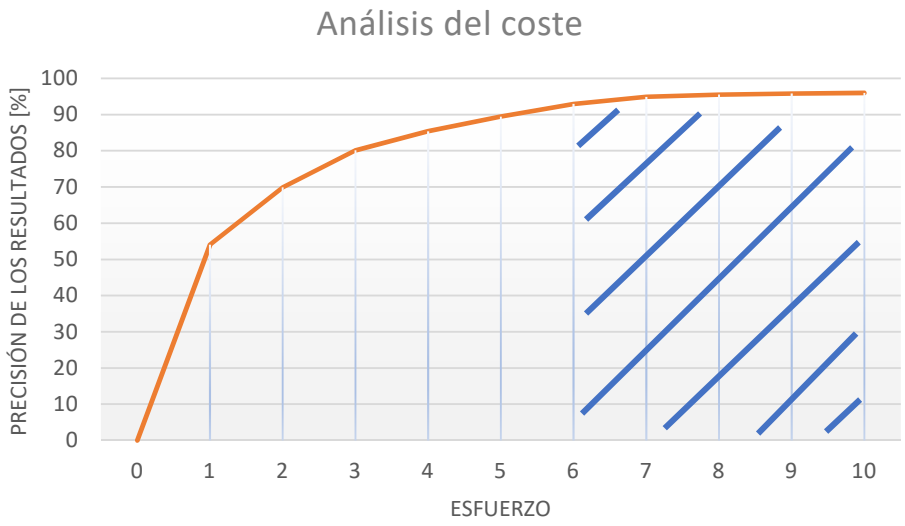


Figura 57: Esfuerzo vs Precisión

En la figura anterior se pretende dar una idea del esfuerzo que supone mejorar un modelo cuando ya ronda una buena precisión. En el eje horizontal se muestra una escala orientativa del esfuerzo que supone mejorar el modelo (0 = Muy poco, 10 = Mucho) y en el eje vertical la precisión que se obtiene en los resultados. Por lo tanto, podemos leer la gráfica como que con poco esfuerzo, al principio conseguimos grandes resultados y que si queremos obtener un resultado muy preciso implica mucho esfuerzo y tiempo. A su vez, dedicarle tanto esfuerzo cuando ya se trata de un modelo preciso, implica mucha dedicación para mejoras pequeñas. Por lo tanto, para según que aplicaciones, es conveniente conformarse con ese punto en el que mejorar más supone una gran inversión de tiempo y cose computacional para obtener un poco más de precisión.

Capítulo 4

Conclusiones y Trabajo Futuro

Llegados a este punto, y como se esperaba, podemos decir que las redes neuronales son un éxito respecto a los modelos ya declarados de *Machine Learning*. La gran flexibilidad que permiten estas arquitecturas, hace posible adaptarse y resolver cualquier problema con un buen ajuste. Y es por ello que dar el salto a redes neuronales ha supuesto una mejoría notable en el resultado.

Hemos conseguido analizar el rendimiento de los modelos desde distintos puntos de vista, viendo un conjunto de herramientas para ello. Y como hemos visto, no podemos fiarnos de una única medida, es importante tener más referencias para evaluar el resultado en su conjunto.

El hecho de trabajar con redes neuronales ha complicado la tarea por las numerosas posibilidades en parámetros y tipos de redes a usar, y en consecuencia sus múltiples combinaciones factibles. Como se ha visto, actualmente muchos modelos se afinan a través del proceso un poco a ciegas, mediante prueba y error, con algunos recursos, pero sobre todo, tiempo.

Sin embargo, esto tiene su recompensa; invertir un tiempo en crear un buen modelo y entrenarlo permite hacer buenas predicciones, lo cual puede ser de gran valor para aquellos profesionales del marketing que quieran conocer los movimientos del mercado futuro entre otros. Pese a que el conjunto de datos recopilado para este trabajo no ha sido muy voluminoso, implementar técnicas predictivas de manera supervisada ha permitido demostrar que nos podemos acercar bastante a las observaciones reales. Hacer el estudio del mercado de esta manera supone un gran ahorro en tiempo y esfuerzo para el ser humano, evitando un proceso farragoso que con las herramientas adecuadas se puede hacer más eficazmente y obviando que con un gran conjunto de datos no seríamos capaces de procesar la información disponible.

No hay que olvidar la importancia que supone trabajar el *data frame* original para obtener el deseado antes de alimentar los modelos. Seguramente nunca recibiremos un conjunto de datos modelado de tal manera que podamos aplicar esos datos al algoritmo directamente, por lo que será necesario hacer una serie de pasos antes para obtener el *data set* de interés.

Como avanzaba al final del capítulo 3, este trabajo muestra el fruto de sacar valor a unos pocos datos. No cabe la menor duda que almacenar y trabajar con mayor cantidad de datos recompensa muy positivamente siempre que se disponga de entornos de ejecución capaces de llevar a cabo estas analíticas (cosa que se ha corroborado con el último algoritmo el cual se ha alimentado de más ejemplos). Además, otro indicio que nos ha intuido la necesidad de más datos ha sido el hecho de ver que todos los modelos han fallado en ciertas fechas por igual; exactamente aquellas en las que por costumbre deberían tomar un valor y ha sido uno inesperado. Quiere decir que si los algoritmos hubieran tenido más ejemplos de estas anomalías seguramente las hubieran podido predecir bien, lo cual implica la necesidad de más datos para tener consciencia de estos singulares sucesos.

Empresas que han tenido al alcance capacidad computacional y datos, han demostrado el potencial que se puede llegar a obtener con algunas aplicaciones de *Machine Learning*, lo cual ha despertado gran interés en muchas otras empresas haciendo que se sumen a esta nueva ola. Es por ello que a día de hoy las empresas se ven casi obligadas a tener que almacenar sus datos para sacarles partido ya que muchas ya hacen uso de ellos, creando organizaciones muy competitivas.

Por lo que ya se intuye que el mercado laboral en un futuro será distinto, mostrando interés por nuevos perfiles que tengan conocimientos sobre el aprendizaje automático. Además, teniendo en cuenta que esto solo es el principio, de cara a un futuro sería interesante seguir conectado a esta tecnología la cual está avanzando a pasos agigantados para ver hasta dónde puede llegar, explorando más allá de este trabajo.

Bibliografía

[1] Artun, O., Levin, D. (2015). *Predictive Marketing: Easy Ways Every Marketer Can Use Customer Analytics and Big Data*. Kindle Edition.

[2] S. Marsland. (2009). *Machine Learning: An Algorithmic Perspective*. CRC Press.

[3] Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. The MIT Press.

[4] Tedesco, B. G. (2003). *A beginner's guide: Artificial Intelligence*. Oneworld Publications.

[5] Rashid, T. (2016). *Make Your Own Neural Network*. CreateSpace.

[6] Yuang, Y. (1999). *Step-sizes for the gradient method*. Providence, RI: American Mathematical Society.

[7] Borovicka, T., Jirina Jr., M., Kordik, P. and Jirina, M. (2012). *Selecting Representative Data Sets*. IntechOpen.

[8] Arias, M., Arratia A. and Xuriguera R. (2012). *Forecasting with Twitter Data*. Computer Science Department - UPC.

[9] Hall, C. (1992). *Neural Net Technology-Ready for Prime-Time*. IEEE Expert.

[10] TensorFlow Official Website: https://www.tensorflow.org/.

[11] Hope, T., Resheff Y and Lieder I. (2017). *Learning TensorFlow*. O'Reilly.

[12] Torres, J. (2016). *Hello World en TensorFlow*. WATCH THIS SPACE Collection.

[13] Langley, P. (1996). *Elements of MACHINE LEARNING*. Morgan Kaufmann Publishers.

[14] Lutz, M. (2013). *Learning Python*. O'Reilly Media.

[15] Garreta, R and Moncecchi, G. (2013). *Learning scikit-learn: Machine Learning in Python*. Packt Publishing.

[16] Matloff, N. (2017). *Statistical Regression and Classification: From Linear Models to Machine Learning*. Chapman & Hall/CRC Texts in Statistical Science.

[17] Rokach, L. and Maimon, O. (2014). *Data mining with decision trees*. World Scientific.

[18] Yi, Z. and Tan, K. (2004). *Convergence Analysis of Recurrent Neural Networks*. Springer Science+Business Media.